StopWatch: A Module for Portable Measurement of Execution Time

William F. Mitchell *
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899 USA

March 28, 1997

Abstract

StopWatch is a Fortran 90 module for portable measurement of execution time. It supports four clocks – wall clock, CPU clock, user CPU clock and system CPU clock – and returns all times in seconds. It provides a simple means of determining which clocks are available, and the precision of those clocks. StopWatch is used by instrumenting your code with subroutine calls that mimic the operation of a stop watch. It supports multiple watches, and provides the concept of watch groups to allow functions to operate on multiple watches simultaneously. This paper describes the software design and use of StopWatch.

1 Introduction

StopWatch is a Fortran 90 module for measuring execution time of program segments. Measuring execution time is an important part of software development, especially for benchmarking and performance tuning. Early versions of Fortran did not support the measurement of execution time, except through non-portable vendor extensions. Fortran 90 [1] introduced a subroutine for measuring wall clock time, but overlooked the more desirable CPU time. It is anticipated that the next Fortran standard, Fortran 95 [5], will include a CPU time subroutine, but it does not break the time into "user" and "system" time like many CPU clock routines, and the standard still does not guarantee that either the wall clock or CPU clock routines will necessarily contain clock information. Moreover, direct use of the routines can be unwieldy, requiring multiple variables to keep track of returned values, differencing the returned values, and conversion of the values to useful units.

StopWatch is designed to be a portable means of measuring execution time. It supports the wall clock, CPU clock, a breakdown of the CPU clock into user and system times, and returns all times in seconds. It provides a simple means of determining which clocks are available, and the precision of those clocks. It is written in a style that allows it to be used with the subset languages F [2] and Elf90 [3], as well as the full Fortran 90 and Fortran 95 languages.

STOPWATCH is used by instrumenting your code with subroutine calls that mimic the operation of a stop watch. The primary routines are **start_watch**, **stop_watch**, **reset_watch**, **read_watch** and **print_watch**. STOPWATCH supports multiple watches, and provides the concept of watch groups to allow functions to operate on multiple watches simultaneously.

The terms *watch* and *clock* have very distinct meanings within STOPWATCH. A *watch* is a variable that is operated on by the STOPWATCH routines. The user can declare as many watches as desired, and operate on them individually or in groups. Each watch contains four *clocks*: wall, CPU, user and system. Within each watch the user can select which clocks to operate on and which clocks to ignore.

Section 2 describes the user interface of STOPWATCH and how it can be used. In Section 3 the design of STOPWATCH is described, including the key Fortran 90 features used and its relationship to object oriented techniques. In this paper, procedure names are shown in **bold face**, derived types in `typewriter type`, variables in *italics*, and Fortran code segments in `typewriter type`.

2 Using STOPWATCH

This section briefly describes the operations that can be performed by the STOPWATCH subroutines. The formal interfaces and detailed descriptions of the routines can be found in the User's Guide [4].

The entities in STOPWATCH that have public accessibility are two derived types and fifteen subroutines. The two derived types are `watchtype`, used for declaring a variable to be a watch, and `watchgroup`, used for declaring a variable to be a handle for a group of watches. These two types have public accessibility, but the internals of the type are private. Any operations performed on a variable of one of these types must be performed by one of the STOPWATCH subroutines.

A watch is a variable declared to be of type `watchtype`. It can be passed to subroutines as an actual argument or through modules like any Fortran variable, but can only be operated on by the STOPWATCH subroutines. Watches must be created by subroutine **create_watch** before they are used, and should be destroyed by subroutine **destroy_watch** when no longer useful to avoid memory leaks. Watches can optionally be given a name through an optional character argument, *name*, in **create_watch**. This name is used in error messages and **print_watch** to identify the watch in the printed output.

Different applications demand different definitions of "time". STOPWATCH supports four clocks in each watch, with each clock measuring a different concept of time. All of them measure time in seconds. The wall clock measures elapsed real time. The cpu clock measures the amount of time the CPU is busy. The CPU time can be further broken down into user and system time.

It is not required that all clocks be used. A watch can be created with any combination of the four clocks. You can also specify a set of default clocks to be used whenever the clocks are not explicitly determined.

Since Fortran 90 does not contain an intrinsic function for CPU time, the implementation of the cpu, system and user clocks is system dependent. Some implementations may support only the cpu and wall clocks, and some implementations may support only the wall clock. This may stabilize as the Fortran 95 intrinsic subroutine **cpu_time** becomes widely available, but system dependent routines for providing CPU time might still be used for the user and system clocks. Clock availability can be determined by subroutine

inquiry_stopwatch.

StopWatch supports multiple watches simultaneously. Often it is useful to perform the same operation on several watches. This is very useful in **pause_watch** and **end_pause_watch** and is convenient for procedures like **read_watch**, **print_watch** and **reset_watch**. To facilitate this, StopWatch supports the concept of watch groups. When calling a StopWatch subroutine, a watch group can be specified instead of a watch. The group is referenced by a variable of type `watchgroup`. Watch groups must be created by **create_watchgroup** before they are used, and should be destroyed by **destroy_watchgroup** when no longer useful to avoid memory leaks. The watches themselves are not destroyed, only the grouping of them. Watches are added to and removed from watchgroups by subroutines **join_watchgroup** and **leave_watchgroup**.

Most StopWatch subroutines take *watch* as the first dummy argument, and accept several forms of *watch*. The forms are:

- `type (watchtype) watch` – a single watch
- `type (watchtype) watch(:)` – an array of watches
- `type (watchgroup) watch` – a watch group handle

In most StopWatch routines, *watch* is an intent(IN) argument, so an array of watches can be specified by an array constructor in the calling statement, for example:

```
type (watchtype) watch :: w1, w2, w3
call print_watch( (/w1,w2,w3/) )
```

Most StopWatch subroutines take *clock* as the (optional) second dummy argument to determine which of the four clocks will be effected by the action. *clock* can be one of the character strings "user", "sys", "cpu", or "wall", or can be an array of such character strings to specify more than one clock. Since *clock* is always intent IN, an array of clock types can be built with an array constructor.

StopWatch is used by inserting subroutine calls into your program. These subroutine calls correspond to the actions performed with a common stop watch. The basic operation of a watch involves starting it, stopping it, resetting it's value to 0, and observing its value. The corresponding StopWatch procedures are:

- **start_watch** – starts an idle watch,
- **stop_watch** – stops a running watch,
- **reset_watch** – sets the clocks on a watch to 0.0,
- **read_watch** – returns the current clock value of a watch, and
- **print_watch** – prints the current clock value of a watch to an output device.

The first argument of **read_watch** is an intent(out) variable in which the clock value is returned. It is either a scalar, a pointer to an array of rank one, or a pointer to an array of rank two depending on whether *watch* and *clock* are scalars or arrays.

When measuring CPU time, it is often desirable to not include the time used by certain parts of the code, such as printing or graphics. In a subroutine, you might not know which watches have clocks that are currently running, so you can not simply stop them before the I/O and start them up again after the I/O. For this, StopWatch provides the pause operation through subroutines **pause_watch** and **end_pause_watch**. When a watch is passed to **pause_watch**, any clocks that are running are placed in a "paused state". A subsequent call to **end_pause_watch** with that watch as an argument starts them running again. A watchgroup containing all of a program's watches can be used to pause everything simultaneously.

Subroutine **option_stopwatch** is provided to set several options with StopWatch, and subroutine **inquiry_stopwatch** is provided to determine the current value of these options and system dependent values of the implementation. All arguments to these subroutines are optional to allow for selective setting/inquiry, and are given reasonable default values. The options that can be set by **option_stopwatch** and read by **inquiry_stopwatch** are:

- *default_clock* – Specifies one or more clock types to be used as the default clocks when the clock argument is omitted.
- *io_unit_print* – Specifies an I/O unit for printed output from routine **print_watch**.
- *io_unit_error* – Specifies an I/O unit for printed error messages.
- *print_errors* – Flag to specify whether or not error messages should be printed.
- *abort_errors* – Flag to specify whether or not the program should abort on an error.
- *print_form* – Specifies the form (seconds or hours:minutes:seconds) for printing time in **print_watch**.

In addition, **inquiry_stopwatch** takes the logical arguments *cpu_avail*, *user_avail*, *sys_avail* and *wall_avail* to indicate the availability of clocks, and the real arguments *cpu_prec* and *wall_prec* to indicate the precision (or resolution) of the clocks.

Figure 1 illustrates a typical use of StopWatch. In this program the main loop is timed using the cpu clock if it is available, and the wall clock if it is not. There are two watches. One measures the total time and the other measures the time for each pass through the loop. **pause_watch** is used to omit the time for printing from the total time.

3 Software Design

StopWatch uses several features of Fortran 90 to achieve a flexible user interface. These include: modules, derived types, public/private accessibility, generic interfaces, and optional arguments. This section describes how these features are used in StopWatch to produce a powerful, flexible, robust module.

StopWatch is designed using an object oriented concept of data abstraction. The default accessibility of module entities is private. Only those entities of use to the user are explicitly given the public attribute. The public entities are two data types, `watchtype` and `watchgroup`, and 15 procedures that operate on those data types.

While the data types themselves are public, the components of the data types are private. This means that users can declare a variable to be of type `watchtype`, but cannot directly access the components of the variable. Any operations on such a variable must occur through the public module procedures. The advantage of this is that upgrades to the StopWatch module can include changes in the definition of the derived types without effecting the user's

```
program sw_example

! declarations
use stopwatch
type(watchtype) :: lap_time, total_time
integer :: i
logical :: have_cpu

! set the clock to use
call inquiry_stopwatch(cpu_avail=have_cpu)
if (have_cpu) then
   call option_stopwatch(default_clock='cpu')
else
   call option_stopwatch(default_clock='wall')
end if

! create and start the watches
call create_watch(lap_time,name='lap')
call create_watch(total_time,name='total')
call start_watch((/lap_time, total_time/))

do i=1,10

! reset the lap timer to 0.
   call reset_watch(lap_time)

! code to be timed goes here

! print the lap time
   call pause_watch(total_time)
   call print_watch(lap_time)
   call end_pause_watch(total_time)
end do

! print the total time
call stop_watch(total_time)
call print_watch(total_time)

! destroy the watches
call destroy_watch(lap_time)
call destroy_watch(total_time)

end program sw_example
```

Figure 1: Example of using StopWatch

code. It also improves the robustness of the module by preventing users from changing the values in the components, which could have unpredictable results.

The StopWatch design also uses the function overloading capabilities of Fortran 90, in which several different types can be passed to the same argument of what appears to be the same procedure. The public module procedures are not procedures at all, but are generic interfaces for several *specific* subroutines that perform the same function on different data types. To present a uniform interface to the user, all forms of the specific routine have the same dummy argument list, except for the types and ranks of the arguments. The same dummy argument list is used uniformly over most of the public procedures, which reduces the number of times the user must refer to the documentation. Additionally, the use of optional arguments allows the user to provide only those arguments that must be specified in the current context. This provides a very uniform yet flexible user interface resulting in a powerful, easy to use module.

When a StopWatch procedure is called, the generic interface determines which specific routine matches the actual argument list. The specific routine converts the arguments to a more general form, supplies default values for the omitted optional arguments, and calls a master version of the procedure which performs the operation.

Consider in detail the procedure **start_watch**. (Since most of the procedures take the same argument list options, other procedures are handled in the same way.) The first two arguments are *watch* and *clock*. As stated in Section 2, *watch* can be a variable of type `watchtype`, a rank one array of type `watchtype`, or a variable of type `watchgroup`, and *clock* is an optional argument which, if present, is either a character variable or an array of character variables.

start_watch is a generic interface to six specific routines. These routines have the names **start_watch_***xy* where *x* is **s**, **a**, or **g** depending on whether *watch* is a scalar, array, or watchgroup, and *y* is **s** or **a** depending on whether *clock* is a scalar or array. The specific routines are private, so they can only be accessed through the generic name. Two of these routines are illustrated in Figure 2 for reference in the following discussion. For conciseness in this illustration, the error checking portions of these routines have been removed.

The worker routine is **start_watch_actual**, which is also private. In this routine, *watch* is an array of type `watch_pointer`, which is a private type containing a pointer to a watch, and *clock* is an array of character variables.

In those specific routines where *clock* is an array, it is a required argument and is simply passed on to the worker routine. In the other specific routines, it is an optional scalar. If it is present, then it is promoted to an array using array constructor notation. If it is not present, then a module array variable containing the default clocks is passed to the worker. This is illustrated in subroutine **start_watch_ss** in Figure 2.

The promotion of *watch* is a little more involved. The worker routine accepts an array of type `watch_pointer` which is not the type accepted by any of the specific routines, but which can easily be constructed from the *watch* argument of each of them. The public and private derived types in StopWatch facilitate this construction. A variable of type `watchtype` does not contain the actual watch data, but instead has a single component which is a pointer of type `watch_actual` which contains the actual data. `watch_pointer` also contains a pointer of type `watch_actual`. The type `watch_group` is a linked list of pointers of type `watch_actual`.

In the routines where *watch* is a scalar of type `watchtype`, there is a local variable, *watches*, that is an array of size 1 and type `watch_pointer`. The pointer in *watches* is set to the pointer in *watch*, and is passed to the worker. This is illustrated in **start_watch_ss**

```
subroutine start_watch_ss(watch,clock,err)
type (watchtype), intent(in) :: watch
character(len=*), optional, intent(in) :: clock
integer, optional, intent(out) :: err
type (watch_pointer) :: watches(1)
watches(1)%ptr => watch%ptr
if (present(clock)) then
   call start_watch_actual(watches, (/clock/),err)
else
   call start_watch_actual(watches,default_clocks,err)
end if
return
end subroutine start_watch_ss

subroutine start_watch_aa(watch,clock,err)
type (watchtype), intent(in) :: watch(:)
character(len=*), intent(in) :: clock(:)
integer, optional, intent(out) :: err
type (watch_pointer), allocatable :: watches(:)
integer :: i
allocate(watches(size(watch)))
do i=1,size(watch)
   watches(i)%ptr => watch(i)%ptr
end do
call start_watch_actual(watches,clock,err)
deallocate(watches)
return
end subroutine start_watch_aa
```

Figure 2: Two of the specific forms of **start_watch**

in Figure 2. In the routines where *watch* is an array of type `watchtype`, the local variable *watches* is an allocatable array. It is allocated with `size(watch)`, the pointers are set to those in *watch*, and it is passed to the worker array. This is illustrated in subroutine **start_watch_aa** in Figure 2. The situation is similar where *watch* is of type `watchgroup`, except a component of `watchgroup` gives the allocation size, and the linked list in `watchgroup` is traversed to set the pointers.

4 Performance

StopWatch has been tested with several compilers and systems to demonstrate its portability. The StopWatch module and six example programs run unchanged on all of these systems, except for a few minor changes in the F version due to differences between the F and Elf90 subsets available at the time of these tests.

Table 1 lists the configurations that have been tested, along with some information about the performance of StopWatch on that system. The computer, operating system, and compiler are listed in the first three columns.

The measurement of CPU time uses a system dependent routine, which is listed in the fourth column. **etime** is a routine that is found on many UNIX systems. The Cray has its own CPU time routine called **second**. On some systems a C routine is used to access the C function **times**. **cpu_time** is the Fortran 95 intrinsic function. Systems labeled "null" do not support a CPU clock.

The fourth and fifth columns provide the precision (or resolution) of the wall and CPU clocks, respectively. The sixth column contains the approximate overhead of StopWatch calls. It was determined by measuring the time required for several thousand calls to **start_watch** and **stop_watch** with a single watch as the argument. The CPU clock was used when available; the wall clock was used on systems that do not support the CPU clock. The overhead and precision together provide an indication of the shortest period of time that StopWatch can reliably measure. A good rule of thumb is $10 * max\{precision, overhead\}$.

The systems for which the CPU routine is "null" support only the wall clock. Those that use **cpu_time** or the Cray routine support only the wall and cpu clocks. All other systems support all four clocks.

5 Acknowledgments

I would like to thank:
Ron Boisvert and Roldan Pozo for many helpful suggestions.
Eite Tiesinga, Karin Remington, Walt Brainard, Neil Campbell, Neil Carlson, Jeroen Groenenboom, Alan Hoffman, Steve Lionel, Christian de Polignac, Mitsu Sakamoto, David Vallance, and Mike Vermeulen for running tests or otherwise providing assistance.

6 Disclaimers

The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology. All trademarks mentioned herein belong to their respective owners.

Table 1: Systems on which StopWatch has been tested.

computer	OS	compiler	CPU routine	cpu precision	wall precision	overhead
Cray C90/6256	UNICOS 8.0.3.2	CF90 1.0.3.5	Cray	4.E-6	4.E-9	2.E-5
DEC AlphaServer 2100 5/250	Digital UNIX V4.0	Digital Fortran 90 V4.1	etime	1.E-3	1.E-4	3.E-5
DEC AlphaServer 2100 5/250	Digital UNIX V4.0	Digital Fortran 90 V4.1	cpu_time	4.E-3	1.E-4	3.E-5
HP 9000/710	HP-UX 10.10	HP Fortran 90 1.0	etime	1.E-2	1.E-3	1.E-4
HP 9000/735	HP-UX 10.20	NAGWare F90 2.2 (284)	etime	1.E-2	1.E+0	4.E-5
IBM RS/6000	AIX 4.1	XLF 4.1	etime	1.E-2	1.E-2	5.E-5
IBM RS/6000	AIX	Imagine1 F Compiler	etime	1.E-2	2.E-2	N/A
Pentium Pro	Windows NT 3.51	Lahey Elf90 v. 2.00c	null	N/A	1.E-2	1.E-3
Pentium 90	Windows NT 3.51	Salford FTN90 V2.15	null	N/A	1.E-3	1.E-1
PowerMac 9500/120	MacOS V7.5.3	Absoft F90 V1.0	null	N/A	1.E-6	5.E-5
Sun SPARCstation 10	Solaris 2.3	Fujitsu Fortran 90 2.03	etime	1.E-2	1.E-3	1.E-4
Sun SPARCstation 10	SunOS 5.4	Sunsoft F90 1.1	etime	9.E-5	1.E-6	2.E-4
Sun SPARCstation 10	SunOS 4.1.3	EPC Fortran 90 V. 1.1.2	etime	1.E-2	1.E-3	1.E-4
Sun SPARCstation 10	SunOS 4.1.3	NAGWare F90 2.1	etime	1.E-2	2.E-2	1.E-4
80486DX-50	Linux 1.2.13	NAGWare F90 2.1	C	1.E-2	1.E+0	6.E-5
80486DX-50	Linux 1.2.13	Imagine1 F Compiler, Rel. 96	C	1.E-2	1.E+0	6.E-5

7 Conclusion

StopWatch is a robust, portable Fortran 90 module for measuring execution time with an easy to use interface. By using the new features of Fortran 90, it is written in an object oriented style that provides a flexible interface and allows for safe upgrades by isolating the user from the details of the data. The StopWatch package is public domain software and can be obtained from `http://math.nist.gov/StopWatch`.

References

[1] J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith and J.L. Wagener, *Fortran 90 Handbook*, McGraw-Hill, 1992.

[2] W. Brainerd, C. Goldberg and J. Adams, *Programmer's Guide to F*, Unicomp, 1995.

[3] Lahey Computer Systems, Inc., *Elf90*, `http://www.lahey.com/elfpage.htm`.

[4] W.F. Mitchell, *StopWatch User's Guide Version 1.0*, NISTIR 5971, 1997, http://math.nist.gov/stopwatch/user_guide.ps.

[5] *Fortran 95 Draft*, `ftp://ftp.ncsa.uiuc.edu/x3j3/`.