UNESCO-NIGERIA TECHNICAL &
VOCATIONAL EDUCATION
REVITALISATION PROJECT-PHASE II

**NATIONAL DIPLOMA IN
COMPUTER SCIENCE**

System Programming

PRACTICALS MANUAL

COURSE CODE: COM 212

Version 1 December 2008

TABLE OF CONTENTS

WEEK ONE

PRACTICALS

LEARNING OBJECTIVES

- **In this Lab we shall be getting familiar with an assembler program and start learning it from scratch. Make sure an assembler is installed in all your systems.**
- **Do not worry even if you are unable to figure out what is happening. Just recognize the format of an assembly program**

SIMPLE ASSEMBLERS

We shall be concerned here with implementing a simple assembler language translator programs. To distinguish between programs written in "assembler code", and the "assembler program" which translates these, we shall use the convention that ASSEMBLER means the language and "assembler" means the translator.
The basic purpose of an assembler is to translate ASSEMBLER language mnemonics into binary or hexadecimal machine code. Some assemblers do little more than this, but most modern assemblers offer a variety of additional features, and the boundary between assemblers and compilers has become somewhat blurred.

A simple ASSEMBLER language

Rather than use an assembler for a real machine, we shall implement one for a rudimentary ASSEMBLER language for the hypothetical single-accumulator machine
An example of a program in our proposed language is given below, along with its equivalent object code. We have, as is conventional, used hexadecimal notation for the object code; numeric values in the source have been specified in decimal.

```
Assembler 1.0 on 01/06/96 at 17:40:45
00 BEG ; count the bits in a number
00 0A INI ; Read(A)
01 LOOP ; REPEAT
01 16 SHR ; A := A DIV 2
02 3A 0D BCC EVEN ; IF A MOD 2 # 0 THEN
04 1E 13 STA TEMP ; TEMP := A
06 19 14 LDA BITS
08 05 INC
09 1E 14 STA BITS ; BITS := BITS + 1
0B 19 13 LDA TEMP ; A := TEMP
12 18 HLT ; terminate execution
13 TEMP DS 1 ; VAR TEMP : BYTE
14 00 BITS DC 0 ; BITS : BYTE
15 END
```

ASSEMBLER programs like this usually consist of a sequence of statements or instructions, written one to a line. These statements fall into two main classes:

Firstly, there are the executable instructions that correspond directly to executable code. These can be recognized immediately by the presence of a distinctive mnemonic for an opcode. For our machine these executable instructions divide further into two classes: there are those that require an address or operand as part of the instruction (as in STA TEMP) and occupy two bytes of object code, and there are those that stand alone (like INI and HLT). When it is necessary to refer to such statements elsewhere, they may be labeled with an introductory distinctive label identifier of the programmer’s choice (as in EVEN BNZ LOOP), and may include a comment, extending from an introductory semicolon to the end of a line.

The address or operand for those instructions that requires them is denoted most simply by either a numeric literal, or by an identifier of the programmer’s choice. Such identifiers usually correspond to the ones that are used to label statements - when an identifier is used to label a statement itself we speak of a defining occurrence of a label; when an identifier appears as an address or operand we speak of an applied occurrence of a label.

The second class of statement includes the directives. In source form these appear to be deceptively similar to executable instructions - they are often introduced by a label, terminated with a comment, and have what may appear to be mnemonic and address components. However, directives have a rather different role to play. They do not generally correspond to operations that will form part of the code that is to be executed at run-time, but rather denote actions that direct the action of the assembler at compile-time - for example, indicating where in memory a block of code or data is to be located when the object code is later loaded, or indicating that a block of memory is to be preset
with literal values, or that a name is to be given to a literal to enhance readability.
For our ASSEMBLER we shall introduce the following directives and their associated compile-time semantics, as a representative sample of those found in more sophisticated languages:

Label	Mnemonic	Address	Effect
not used	BEG	not used	Mark the beginning of the code
not used	END	not used	Mark the end of the code
not used	ORG	location	Specify location where the following code is to be loaded
optional	DC	value	Define an (optionally labelled) byte, to have a specified initial value
optional	DS	length	Reserve length bytes (optional label associated with the first byte)
name	EQU	value	Set name to be a synonym for the given value

Besides lines that contain a full statement, most assemblers usually permit incomplete lines. These may be completely blank (so as to enhance readability), or may contain only a label, or may contain only a comment, or may contain only a label and a comment.

WEEK 2 PRACTICALS

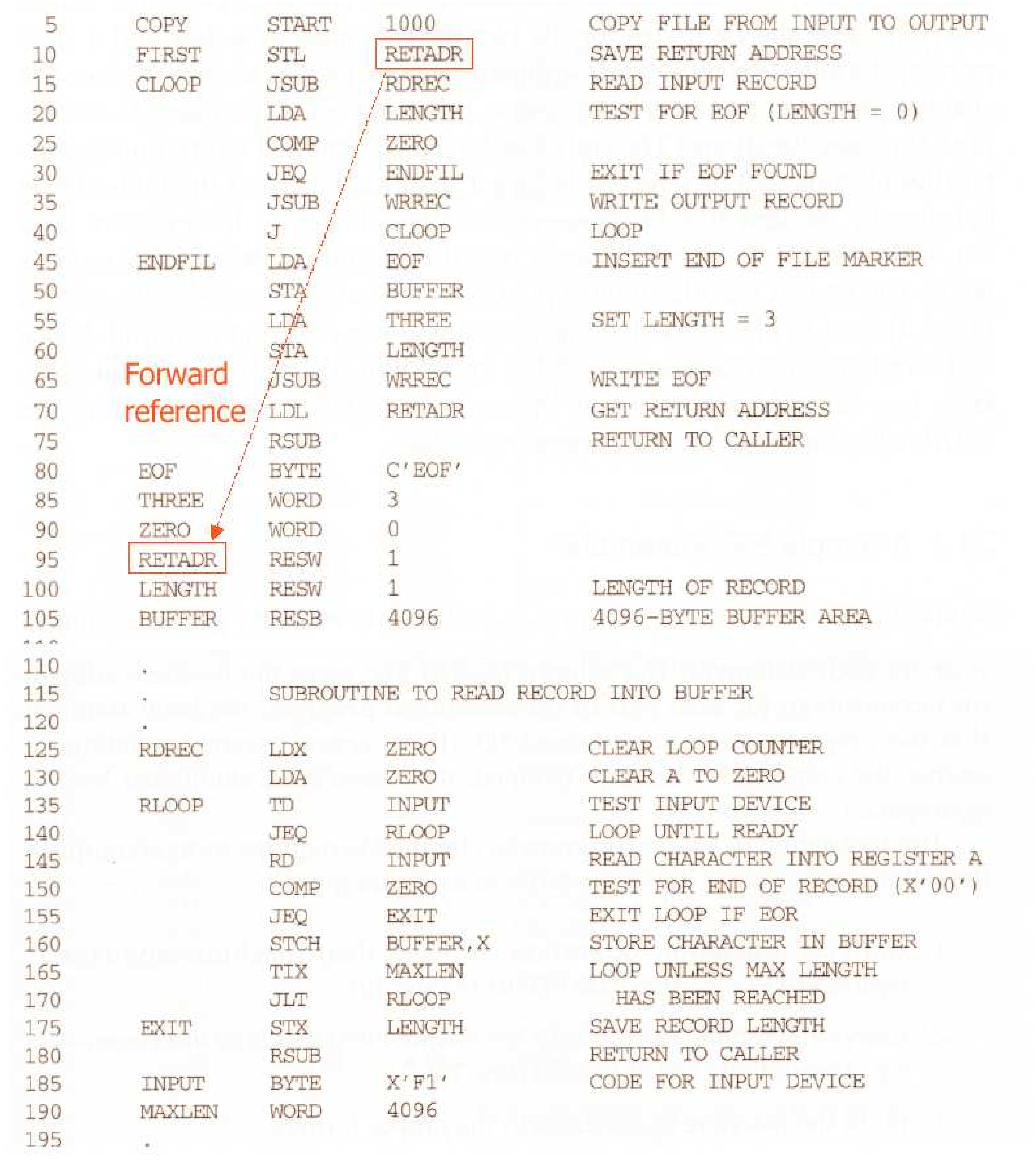
Learning Outcome

THE TEACHER IS STRONGLY ADVICED TO VISIT THE WEB SITES LISTED SO THAT HE CAN BE FAMILIAR TO SOME OF THE CONCEPTS BEFORE ACTUAL PRACTICAL TAKES PLACE

Assembler Programming

NOTE: Some activities may not be completed within the allotted time. You may extend according to the understanding of the students.
Your first Intel Assembler programs
Download some sample Intel programs for practice by your students on this site : http://teaching.idallen.com/dat2343/OOs/
The sidebar contains links to many Intel assembler programs. First is a one-page program named onepage.asm that you can select and download to make sure your assembler and linker are working correctly.
WARNING: (If you can't assemble and run this simple program without any warnings or errors, something is wrong with your assembler or linker!)
The command lines used to assemble and link this program are given in the comments at the beginning of the program. Either put the ASM and VAL programs in the same directory as your program, or adjust your DOS PATH variable to include where they reside. Read the documentation for more details on how the ASM and VAL programs work.

PROGRAM EXAMPLE


Forward reference

```
COPY      START   1000        COPY FILE FROM INPUT TO OUTPUT
FIRST     STL     RETADR      SAVE RETURN ADDRESS
CLOOP     JSUB    RDREC       READ INPUT RECORD
          LDA     LENGTH      TEST FOR EOF (LENGTH = 0)
          COMP    ZERO
          JEQ     ENDFIL      EXIT IF EOF FOUND
          JSUB    WRREC       WRITE OUTPUT RECORD
          J       CLOOP       LOOP
ENDFIL    LDA     EOF         INSERT END OF FILE MARKER
          STA     BUFFER
          LDA     THREE       SET LENGTH = 3
          STA     LENGTH
          JSUB    WRREC       WRITE EOF
          LDL     RETADR      GET RETURN ADDRESS
          RSUB                RETURN TO CALLER
EOF       BYTE    C'EOF'
THREE     WORD    3
ZERO      WORD    0
RETADR    RESW    1
LENGTH    RESW    1           LENGTH OF RECORD
BUFFER    RESB    4096        4096-BYTE BUFFER AREA
.
.         SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC     LDX     ZERO        CLEAR LOOP COUNTER
          LDA     ZERO        CLEAR A TO ZERO
RLOOP     TD      INPUT       TEST INPUT DEVICE
          JEQ     RLOOP       LOOP UNTIL READY
          RD      INPUT       READ CHARACTER INTO REGISTER A
          COMP    ZERO        TEST FOR END OF RECORD (X'00')
          JEQ     EXIT        EXIT LOOP IF EOR
          STCH    BUFFER,X    STORE CHARACTER IN BUFFER
          TIX     MAXLEN      LOOP UNLESS MAX LENGTH
          JLT     RLOOP         HAS BEEN REACHED
EXIT      STX     LENGTH      SAVE RECORD LENGTH
          RSUB                RETURN TO CALLER
INPUT     BYTE    X'F1'       CODE FOR INPUT DEVICE
MAXLEN    WORD    4096
.
```

WEEK 3

PRACTICAL

This Week Practical is the continuation of what we started last week. Make sure you have downloaded at least one Intel Assembler program and get to study it by carefully following the various steps outlined.
Alan's star.asm program (http://teaching.idallen.com/dat2343/00s/stars.asm prints numbers of asterisks based on one key of input, and noecho.asm that loops reading characters without echoing them to the screen.
There is a longer program (mostly comments!) named first.asm http://elearning.algonquincollege.com/coursemat/pinka/dat2343/project/lec_IO.pkg that uses Alan Pinck's I/O Package. (http://teaching.idallen.com/dat2343/00s/stars.asm (This package contains some subroutines for inputting and outputting numbers.) You will need Alan's I/O package in the current directory on your disk to assemble this example. This program is a good test to see if the I/O package downloaded to your computer correctly. Program addtwo.asm is another example that uses the I/O package.

The above programs, if properly downloaded give you a good idea of what an assembler program .

WEEK 4-6

PRACTICAL

For the next Three Weeks, you will again learn to write a simple assembler program step by step and try to understand what it does.

EXAMPLE PROGRAM

Write a program that has no input and outputs a continuous string of numbers, starting with 1, and counting by one. In other words, your output should be 1 2 3 4 5 6 ….

Solution

Step 1: 1 00 We clear the accumulator resetting it to zero and then load the content of memory block 00 into the accumulator. The accumulator now contains 001.

Program Counter	25	Accumulator	1
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02		32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream:

Step 2: 5 00 We output the contents of memory block 00. We do this because we need a 1 as the first number in our output stream. Memory block 00 already contains a 1 so we can output directly from it.

Program Counter	26	Accumulator	1
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02		32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream: 1

Step 3: 2 00 To construct the rest of the output stream we need to increment each output value by 1 to create the next value in the stream. Since we have 1 in the accumulator we need to add another 1 to it to get the next output value, a 2. So in this instruction we add the content of memory block 00, a 1, to the contents of the accumulator without resetting it first. The accumulator content is now 2.

Program Counter	27	Accumulator	2
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02		32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Step 4: 6 02 We want to output the next number of the output stream, 2, which is now in the accumulator. Remember that we cannot output directly from the accumulator so we first need to store its content to a memory block. In this instruction, we select memory block 02 for this purpose. You could have chosen any empty block. So now memory block 02 contains a 2.

Program Counter	28	Accumulator	2
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02	2	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Step 5: 5 02 We can now output the next number of the output stream from memory block 02.

Program Counter	29	Accumulator	2
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02	2	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	

29	502	38	

Output Stream: 1 2

Step 6: 8 27 Since we need to increments again by 1 to get the next output value we can just repeat the increment (block 27)-store (block 28)-output (block 29) sequence of instructions infinitely to construct the rest of the output stream. The first instruction in this sequence, increment by 1, resides in memory block 27. Therefore, that's the location to which we need to jump to repeat the sequence.

Program Counter	30	Accumulator	2
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02	2	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Follow the changes in the next few steps of the program. Note the pattern that develops. I have only shown the increment and store steps (lumped into one figure) to emphasize how the contents of the accumulator and memory block 02 change for every execution of the increment-store-output sequence.

Steps 7 & 8

Program Counter	27	Accumulator	3
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02	3	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream (after step 9): 1 2 3

…… Steps 11 & 12

Program Counter	27	Accumulator	4
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02	4	32	
……		33	
25	100	34	

26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream (after step 13): 1 2 3 4
…… Steps 15 & 16

Program Counter	27	Accumulator	5
Memory Block	Block Content	Memory Block	Block Content
00	1	30	827
01		31	
02	5	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream (after step 17): 1 2 3 4 5
The sequence keeps repeating infinitely.

WEEK   7-9

PRACTICAL

This is another program written in C language. It will help you detect errors. Type it and run it.

PROGRAM

```
#include < stdio.h >
#include < errno.h >

main()
{
  int i;
  FILE *f;

  f = fopen("~lagos/nonexist", "r");
  if (f == NULL) {
    printf("f = null.  errno = %d\n", errno);
    perror("f1");
  }
}
```

ch1a.c tries to open the file ~lagos/nonexist for reading. That file doesn't exist. Thus, fopen returns NULL (read the man page for fopen), and sets errno to flag the error. When you run the program, you'll see that errno was set to 2. To see what that means, you can do one of two things:

1. Look up the errno value in /usr/include/errno.h (You will have to eventually look at /usr/include/sys/errono.h on UNIX flavor machines since on that type of system, the C standard errno.h does have "#include < sys/errno.h >" in it.). You'll see the line:

```
    #define ENOENT          2               /* No such file or directory */
```

2. Use the procedure "perror()" -- again, read the man page. It prints out what the errno means. Thus, the output of f1 is

```
   f = null.  errno = 2
   f1: No such file or directory
```

This is the standard interface for errors.

WEEK 10- 12

PRACTICALS

This practical is based on acquainting the student on the use of DOS DEBUG command and also
Understanding some instructions such as PUSH, POP, CALL, and INT.
The student should make sure that he has visited the following websites before the practicals begins.
http://chesworth.com/pv/technical/dos_debug_tutorial.htm
http://www.datainstitute.com/debug1.htm
Saving output from DEBUG
Some of the assignments require you to save the output of DEBUG in a file. If you're running in a DOS Window under Windows 9x, you can always use the mouse to copy text and paste it into another application such as Notepad, Wordpad, or Write. (Do not save the window as a graphic using Print Screen - the resulting file is huge and unnecessary. Save the text only, using cut-and-paste into another application such as Write, Wordpad, or Notepad, and print from there.)
When you print some screen text or programs, you must use a Courier or Terminal fixed-width font so that the text lines up. Do not print with a variable-width font such as Times, Tahoma, or Arial!
If you're running in pure-DOS mode, without Windows, study the examples below under the heading DEBUG Scripts. You can enter a few DEBUG commands into a text file, and have DEBUG read the file and execute the commands while you redirect the output into an output file. Then, you can print the output files (using a Courier or Terminal fixed-width font).
Understanding PUSH, POP, CALL, and INT
The PUSH, POP, CALL, and INT instructions all use the stack (the memory area pointed at by SS:SP). The following DEBUG scripts and their annotated output files show what happens. Each of the input files was run through DEBUG to produce the corresponding output file, to which explanatory comments were added by hand:

```
C:> debug <push_pop.txt >push_pop_out.txt
C:> debug <call_push.txt >call_push_out.txt
C:> debug <int_push.txt >int_push_out.txt
```

Read these Output files for annotated examples of the workings of PUSH/POP, CALL, and INT: DEBUG Input	Annotated DEBUG Output
push_pop.txt	push_pop_out.txt
call_push.txt	call_push_out.txt
int_push.txt	int_push_out.txt

WEEK 13 – 15

PRACTICALS

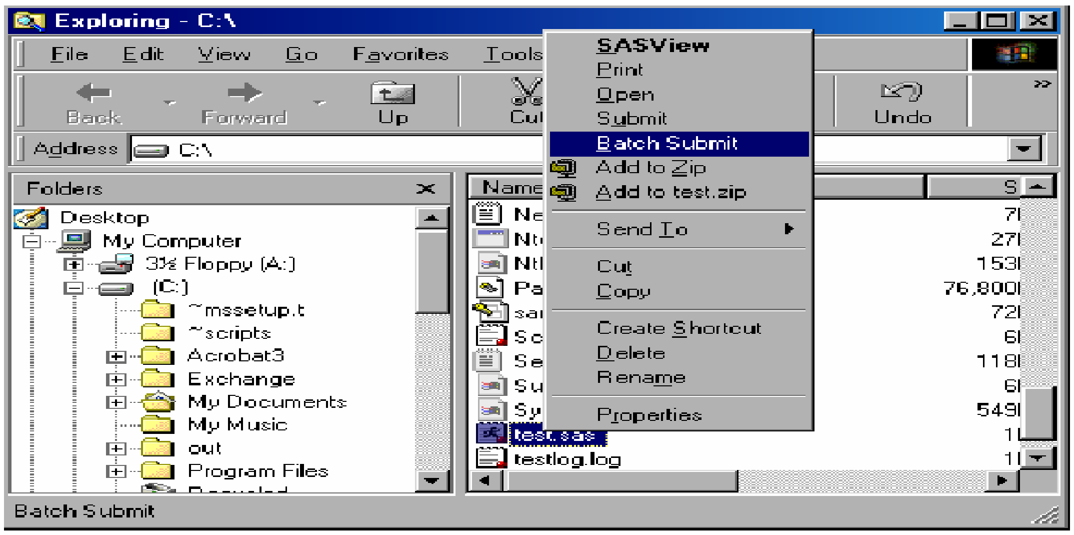
This Practical focuses on Batch processing. It is expected that the students will find great reward in going through examples and practicals.

Examples of Batch Processing under Windows

Several examples of executing SAS batch jobs under Windows 95, 98, NT, ME and Windows 2000 operating systems are demonstrated. These examples will work with all of the operating systems listed. The screen shots for those using Windows 2000, Windows 95/98/ME/XP may be slightly different.
(Note: For all the examples, assume that the SAS System is installed in the "C:\Program Files\SAS Institute\SAS\V8\" directory. Change the path as necessary).

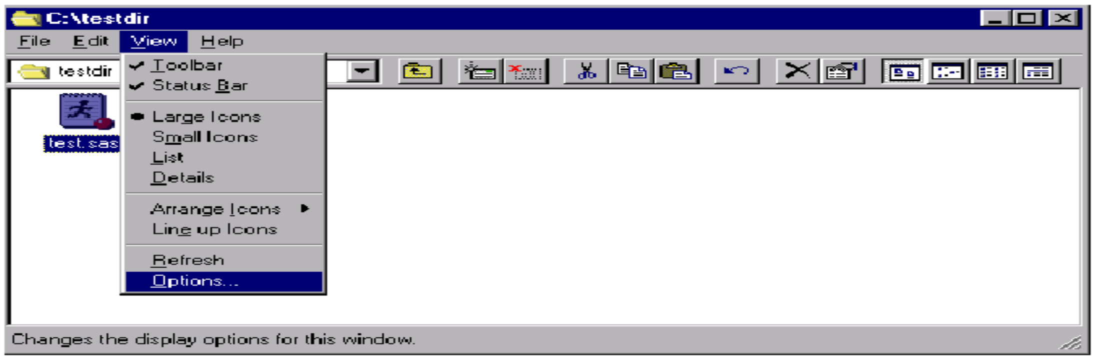
1. The first example of running SAS in batch is the simplest way. If SAS is registered properly,
open Explorer and right click on the program to be executed. (SAS should be registered properly when you install SAS the first time. If SAS is not registered properly refer to the example on the next page.) Select the Batch Submit option. This will execute the program as a batch job. The .LOG and .LST file will be located in the same folder as the program by default.
The .LOG file is the log for the program that has been executed. This file will contain useful information if the program did not run correctly.
The .LST file is the output file for the program that has been executed. This file is only created if
there is output for the program.

Example 1:

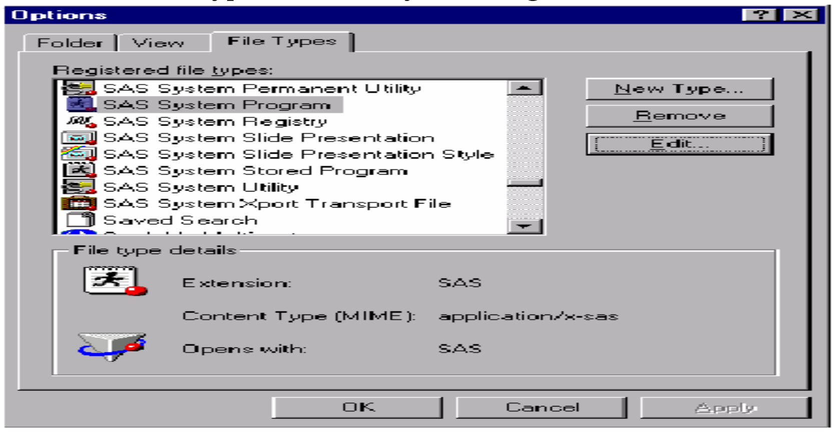


You can also have .SAS files run in batch when you double click on the files within Explorer. The default action for .SAS files needs to be set to Batch Submit.
If the default action for the .SAS files needs to be changed, follow this example:

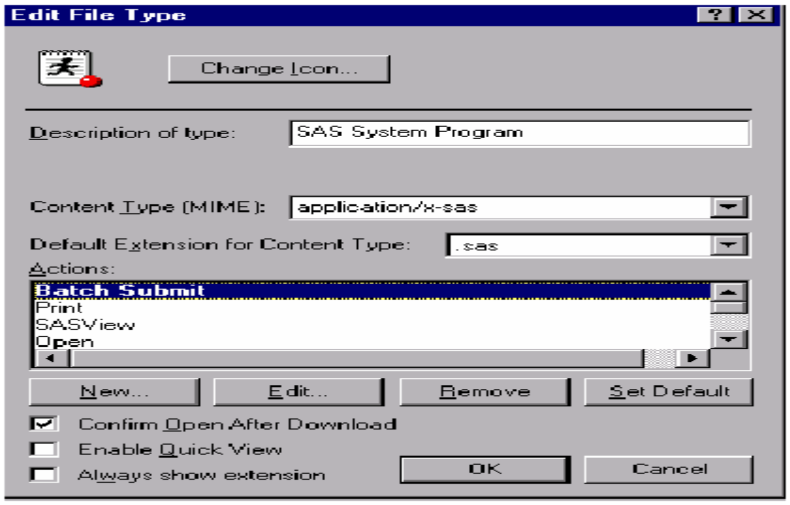
Open up Windows Explorer and Select View > Options.



Now Select >File Types and SAS System Program



Select >Edit


Edit File Type

Description of type: SAS System Program

Content Type (MIME): application/x-sas

Default Extension for Content Type: .sas

Actions:
Batch Submit
Print
SASView
Open

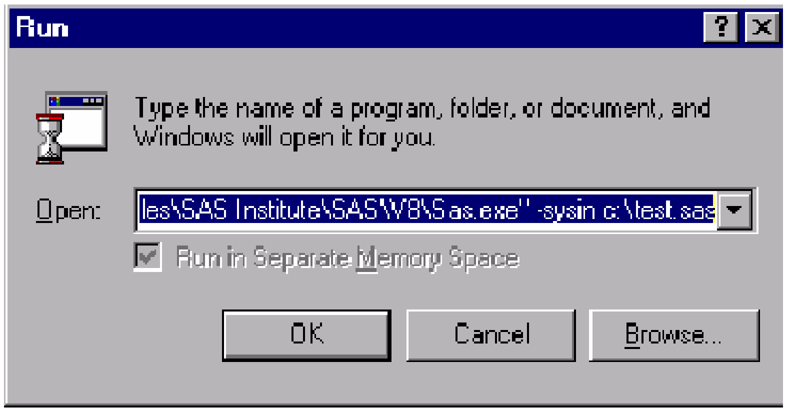
[✓] Confirm Open After Download
[ ] Enable Quick View
[ ] Always show extension

D. Highlight the Batch Submit action and Select > Set Default and then select OK. Once this is set, double clicking a .SAS file will execute the SAS program in batch Mode. The .LOG and .LST files will be created in the same folder as the .SAS file.

2 EXAMPLE 2 .

The second example of SAS batch submissions uses the SAS executable command with the -SYSIN option. Select START>Run and type in the command illustrated in Example 2 below. Specify the location of the program. This example will run the TEST.SAS program located in root of the C drive. The splash screen can be eliminated by using the –NOSPLASH option. Also, adding the –ICON option will minimize the DOS window when the program is started. The .LOG and LST files will reside in the SAS root directory.
The -SYSIN option specifies the SAS program file that will be run in batch. The path needs to be
a valid Windows path.

Example:

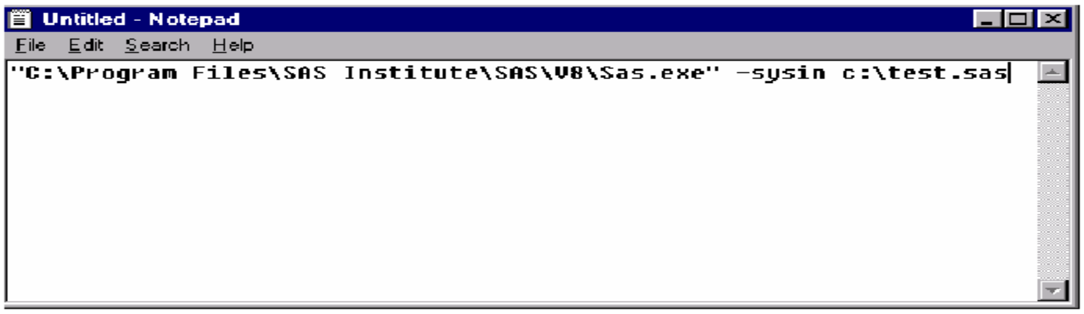

Run

Type the name of a program, folder, or document, and Windows will open it for you.

Open: les\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test.sas

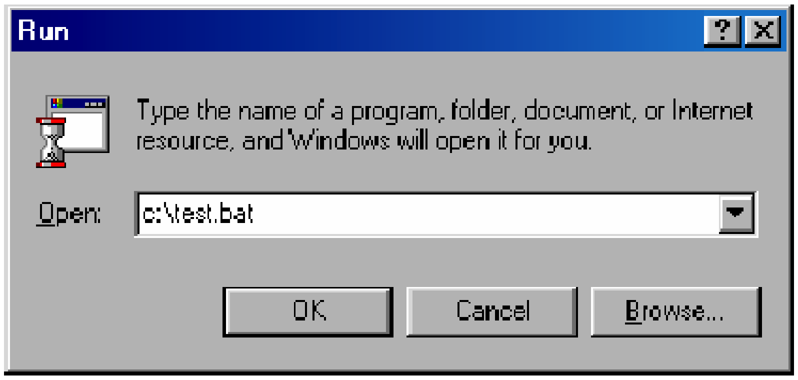
[✓] Run in Separate Memory Space

3. Another option using the command in the previous example is to create a .BAT file. This is a
file that can be executed by the operating system. The easiest way to create a .BAT file is to use
Notepad or another text editor. Here is an example of the text that can be used within the file.



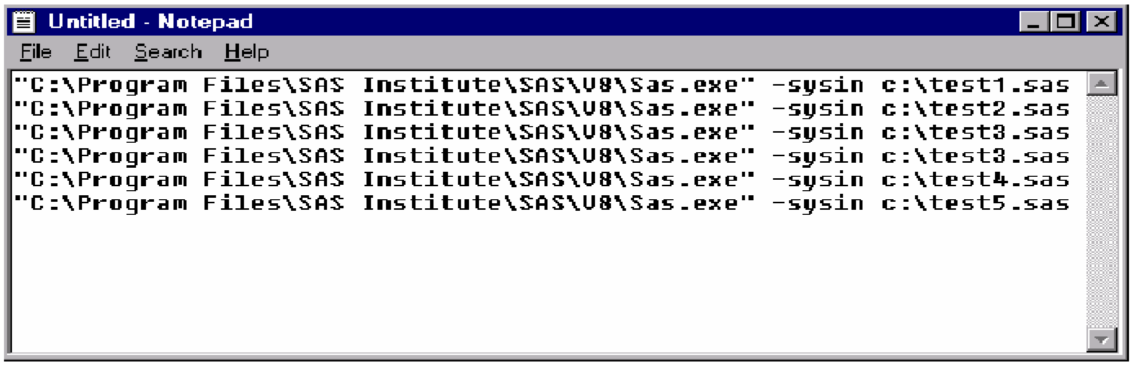
```
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test.sas
```

Once this file is created, select Start>Run, to execute the job or double click on the .BAT file.
The .LOG file and the .LST files will reside in the folder of the .BAT file. The destination of the
.LOG and .LST file can be changed using the –LOG and the -PRINT options.
The –PRINT option is used to change the destination folder for the output of the program.
The –LOG option is used to change the destination folder for the log of the program that has been
executed.
Example: "C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -SYSIN c:\testprog.sas
-NOSPLASH -ICON -PRINT c:\test.lst –LOG c:\test.log
Example:


Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
c:\test.bat
OK
Cancel
Browse...

Running a .BAT program will open a DOS window, which will not close until the job is finish.

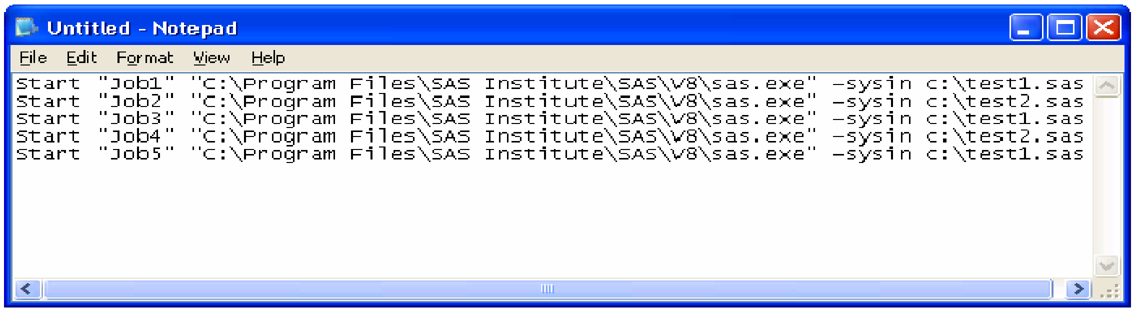
4a. Running more than one job at a time can be done within the .BAT file. Use the previous example but add more programs within the TEST.BAT file. This example will execute 5 jobs concurrently.
Example:



```
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test1.sas
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test2.sas
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test3.sas
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test3.sas
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test4.sas
"C:\Program Files\SAS Institute\SAS\V8\Sas.exe" -sysin c:\test5.sas
```

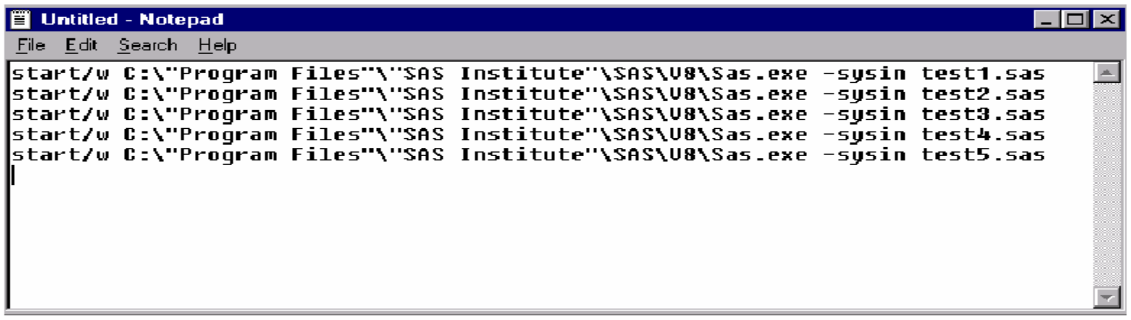
4b.
This example shows how to run concurrent batch jobs for Windows 2000 and Windows XP. The Start command is needed so the batch jobs will run concurrently. The START command also requires a title. The title can be double quotes but this example uses Job#.

Example:



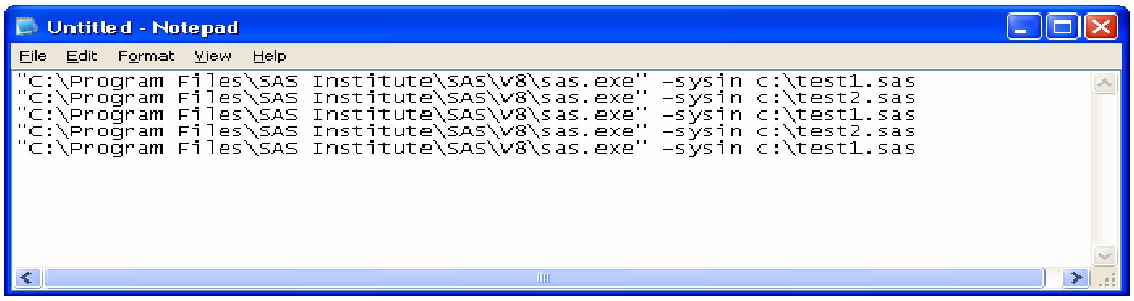
```
start "Job1" "C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test1.sas
start "Job2" "C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test2.sas
start "Job3" "C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test1.sas
start "Job4" "C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test2.sas
start "Job5" "C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test1.sas
```

5a. This next example shows how to create a batch (TEST.BAT) file that will run 5 jobs consecutively, instead of concurrently. The log (.LOG) and output files (.LST) will be located in the directory where the .BAT file is located unless you use the –PRINT and –LOG options on the command line. (This example is for Windows 95,98,ME,NT)
Example:



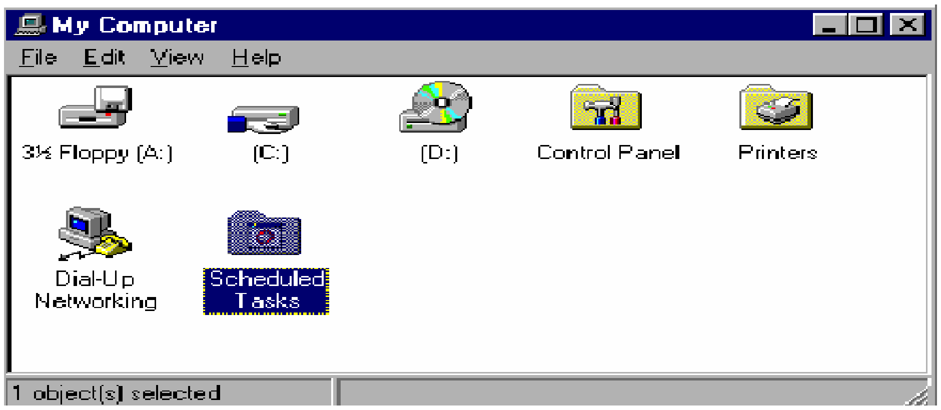
```
start/w C:\"Program Files"\"SAS Institute"\SAS\V8\Sas.exe -sysin test1.sas
start/w C:\"Program Files"\"SAS Institute"\SAS\V8\Sas.exe -sysin test2.sas
start/w C:\"Program Files"\"SAS Institute"\SAS\V8\Sas.exe -sysin test3.sas
start/w C:\"Program Files"\"SAS Institute"\SAS\V8\Sas.exe -sysin test4.sas
start/w C:\"Program Files"\"SAS Institute"\SAS\V8\Sas.exe -sysin test5.sas
```

5b. This example is for Windows 2000 and Windows XP. This example shows how to execute batch jobs consecutively.
Example:



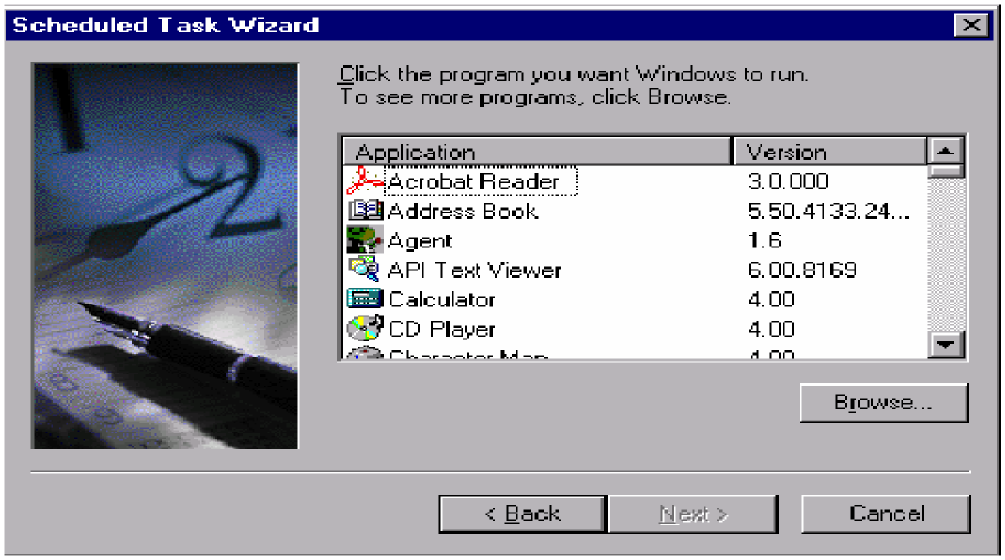
```
"C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test1.sas
"C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test2.sas
"C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test1.sas
"C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test2.sas
"C:\Program Files\SAS Institute\SAS\V8\sas.exe" -sysin c:\test1.sas
```

6. This next section will show examples of using a scheduler to run SAS jobs at a specific time. These jobs can be executed overnight without having anyone at the computer. The Scheduled Tasks program is located under My Computer for Windows NT 4 and is located in the Control Panel under Windows 2000 and Windows 98 and ME. Windows95 doesn't have this scheduler.



1 object(s) selected

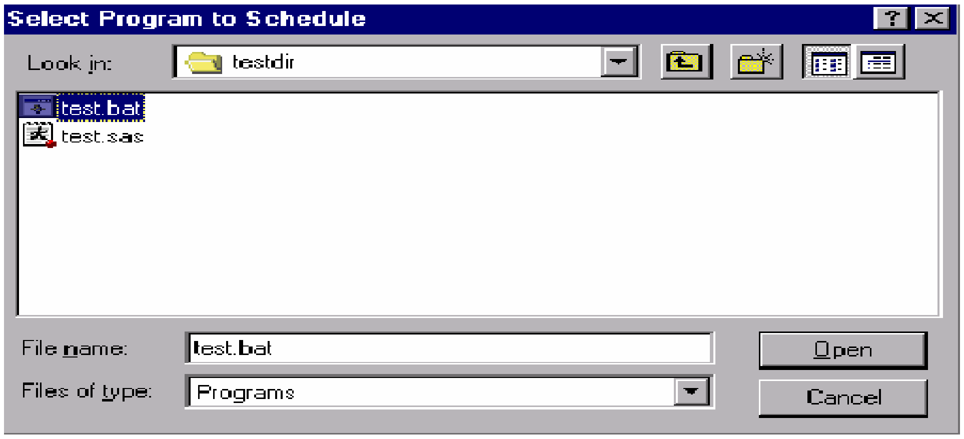
Double click on the Scheduled Tasks icon to invoke the following wizard:



Click the program you want Windows to run.
To see more programs, click Browse.

Application	Version
Acrobat Reader	3.0.000
Address Book	5.50.4133.24...
Agent	1.6
API Text Viewer	6.00.8169
Calculator	4.00
CD Player	4.00

Now that the Scheduler Wizard is running, select the Browse button and point to the TEST.BAT file.

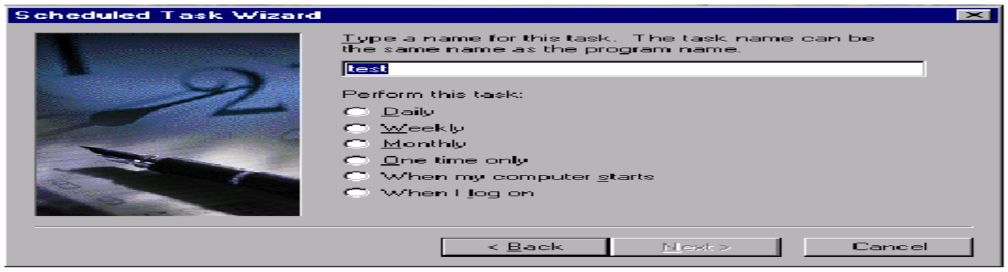


Look in: testdir

test.bat
test.sas

File name: test.bat
Files of type: Programs

Once a file has been selected, choose how often this task should be executed and select the next button.



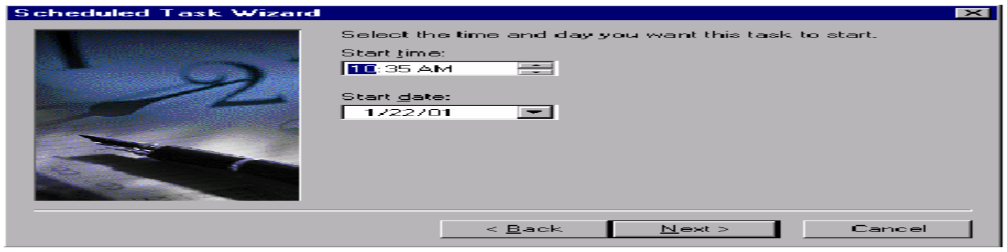
Type a name for this task. The task name can be the same name as the program name.

test

Perform this task:

- Daily
- Weekly
- Monthly
- One time only
- When my computer starts
- When I log on

Now select the date and time this job should be executed and select Next.



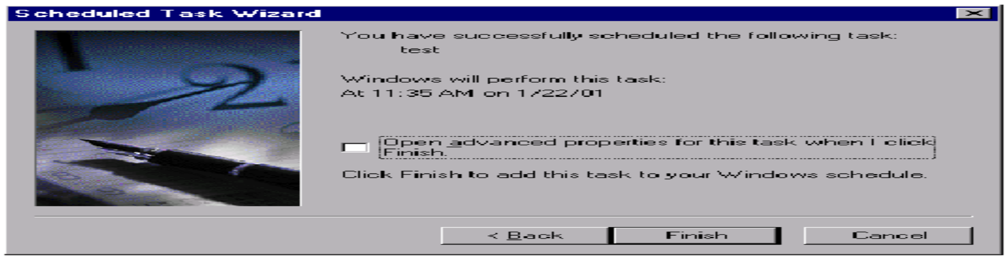
Select the time and day you want this task to start.

Start time:
10:35 AM

Start date:
1/22/01

After the time has been selected, the prompt for the user name and password will appear. Enter your user name and password your job is running under and select Next.

Now that the user name and password are entered, the following message window will be displayed.



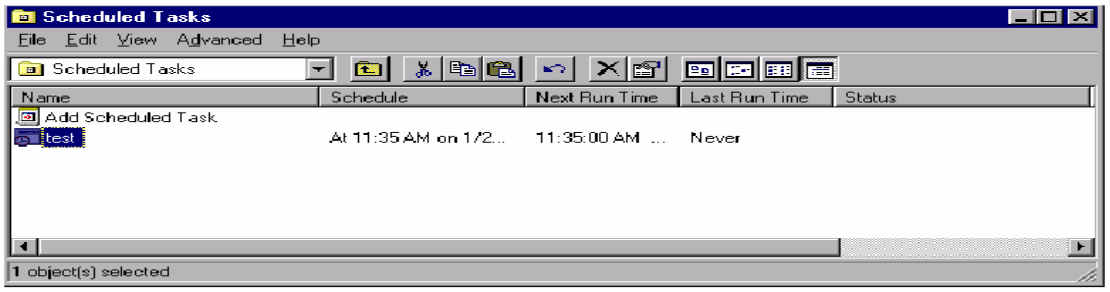
You have successfully scheduled the following task:
test

Windows will perform this task:
At 11:35 AM on 1/22/01

Open advanced properties for this task when I click Finish.

Click Finish to add this task to your Windows schedule.

The task has now been scheduled, select the Finish button.
This will display the Task Scheduler's main menu and the job will be listed there.



Name	Schedule	Next Run Time	Last Run Time	Status
Add Scheduled Task				
test	At 11:35 AM on 1/2...	11:35:00 AM ...	Never	

1 object(s) selected