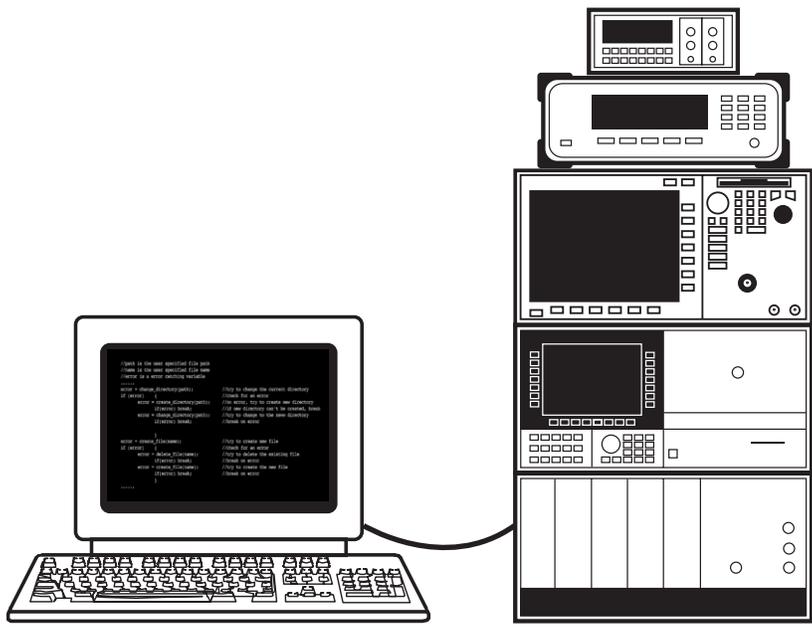
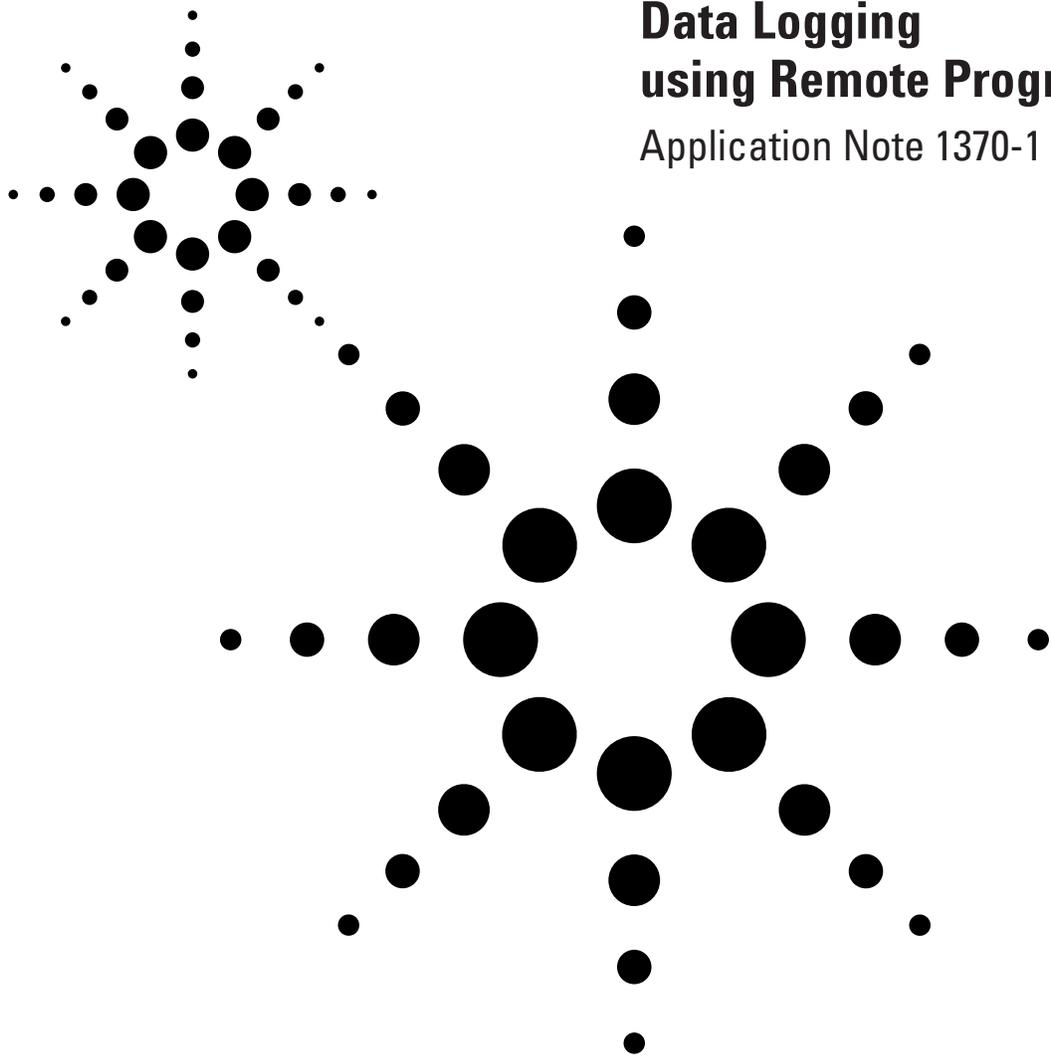


Data Logging using Remote Programming

Application Note 1370-1



Introduction

One of the most obvious and useful applications of remote programming of test instruments is to run the same test periodically and simultaneously, saving the data from each measurement to the computer's hard drive. This allows an instrument to continuously monitor a system without user interaction for hours or even days at a time. For instance, a test could be set up using an Agilent 86100A Infiniium DCA (digital communications analyzer) to measure the extinction ratio every thirty seconds for an hour. Similarly, an Agilent 8614XB optical spectrum analyzer (OSA) could measure signal-to-noise ratio in a DWDM system once an hour for two days. Unfortunately, this is deceptively complex and requires a surprising amount of programming overhead. In addition to the usual programmatic requirements for controlling and communicating with the instrument or instruments, the program must include functions which incorporate user inputs and controls, file set-up and I/O, the delays between measurements, and elementary error handling.

Remote programming is the process of using a PC to issue commands to an instrument and read measurement results back to the PC. This is most commonly done using a General Purpose Interface Bus (GPIB) in accordance with the IEEE 488.2-1992 standard. Agilent manufactures GPIB cards for PC's as well as UNIX® workstations and also Ethernet-to-GPIB gateways. Additionally, virtually all of Agilent's lightwave instruments are equipped

for remote programming via GPIB and come with documentation on the commands and functions available. Almost any command which can be input through the instruments front panel has a remote command equivalent.

The simplest format in which to output data is a delimited ASCII text file. A delimited ASCII text file is a simple text file where a specific character, usually a *tab* or *comma*, separates each datum. Additionally, this type of file offers excellent generality. An ASCII file can easily be transferred to a number of database (such as Oracle® 8 or Microsoft® Access) and spreadsheet programs (such as Microsoft Excel or Lotus® 1-2-3). Most other file formats require a file header which is unique to that file format. In addition to limiting the output file's compatibility to one program, writing the header requires additional program complexity. Many commonly used programming languages (including Microsoft Visual C++ and Visual Basic, National Instruments LabView, and Agilent VEE) include ActiveX controls which allow direct data transfer into database and spreadsheet programs but this approach adds complexity and is again done at the cost of generality. A text file format offers the maximum of simplicity and generality and as such is perfect for example purposes.

Programming Specifics

With the added complexity of elements including user input, file I/O, and error handling, program flow becomes an issue of utmost importance. The first step is to prompt the user for the necessary information such as the delay between measurements, the total test duration, and the name and location of the output file. After the necessary information is received from the user, the program must set up the specified output file and establish communication with the instrument or instruments. After this step, the program enters a loop which is repeated until the specified time limit is reached, an error occurs, or the user aborts the program. With each iteration of this loop, the data is acquired from the instrument, any necessary calculations are performed, the data is formatted and sent to the output file, and the program simply pauses until it is time to repeat the loop. An outline of the program can be seen in Figure 1.

The user input is one of the simpler facets of a data-logging program. Graphical languages like LabView and VEE are designed with user interface as a top priority. In such languages, setting up a user input is as simple as dragging and dropping a few boxes. Even in C++ or Visual Basic, getting a simple input from the user is not difficult. In C++, the simplest way is to prompt the user by printing a text string to the program window, and then looking at the user's response. In Visual C++ or Visual Basic, one can even use "forms" which use familiar MS Windows® controls. One important thing not to overlook, however, is to check for errors in this process. When using pre-

1. Prompt the user for test duration, frequency, output path, etc.
2. Set up the instrument and the output file(s).
3. Perform measurement and gather data from the instrument.
4. Perform running calculations (for example average, minimum, maximum).
5. Send data to the output file (can be performed after the loop).
6. Wait until it is time for the next measurement.
7. Repeat steps 3–6 until the time limit, an error, or asked to stop.

Figure 1. Program Outline

made input functions like those in graphical languages, invalid inputs are usually disallowed, so these types of errors are less of a problem. On the other hand, in a language such as C++ the program may prompt for a length of time and there is nothing to stop the user from typing in his or her name, if for no other reason than to see if the programmer was being careful. Malicious inputs aside, it is quite possible for the user to make a typo or other mistake and the program must be designed to deal with input errors.

The next step is to set up the instrument and the output file. The language and the operating system used determine the complexity of this step. Error catching is vital in this step as the user can input file and path names which may not exist which can cause errors. In MS Windows, for example, an error may be caused if the user specifies a directory that is not present, a file that already exists, or both. It is vital that the program knows what to do in any case. It is simple enough to create a new directory or overwrite an old file, but the program must know how and whether or not to perform these tasks. An

example of what this may look like is shown in Figure 2, page 4. Notice that the program first assumes that the directory exists and that the file does not exist. If there is a problem, it assumes that the source of error was the directory needing to be created or the file needing to be cleared and then the program performs the appropriate task. This very simple error correction will handle the vast majority of errors stemming from the file set-up step. It is also necessary in most languages to set up an output stream to the output file. An output stream is simply memory buffer that holds the data while the file is being written. This usually only requires a few lines of code, but error checking is necessary here as well.

The other part of the set-up step in the program is to set up the instrument. This step includes verifying the communications link and establishing communication with the instrument. This is usually also the appropriate time to change the instrument or instruments to the desired settings. In some cases, it may be necessary to change the settings periodically. For instance, one measurement may be taken in

```

//path is the user specified file path
//name is the user specified file name
//error is a error catching variable
.....
error = change_directory(path);           //try to change the current directory
if (error) {                             //check for an error
    error = create_directory(path);       //on error, try to create new directory
    if(error) break;                     //if new directory can't be created, break
    error = change_directory(path);       //try to change to the neww directory
    if(error) break;                     //break on error
}
error = create_file(name);                //try to create new file
if (error) {                             //check for an error
    error = delete_file(name);           //try to delete the existing file
    if(error) break;                     //break on error
    error = create_file(name);           //try to create the new file
    if(error) break;                     //break on error
}
.....

```

Figure 2. Psuedocode example of error catching for a new file

decibels and another in watts. In such a case the settings would have to be changed between measurements or as part of each measurement process. However, one-time settings changes, such as selecting the output format, should also be done in this step. Again, it is important that the program be equipped to handle common errors from this process and provide the user with information on any errors. For instance, if a communications link cannot be set up, the program should identify that as the problem. It is far easier to find and correct a problem in the system if the program informs the user from where the error originates.

After all of the necessary parameters have been gathered and the set up is complete, the real work of the program can begin. The first step of the measurement

loop, of course, is to actually take the measurements. After the GPIB link has been set up, it is usually just a matter of sending string commands over the link and then reading the response. As usual though, error checking is vital. Any number of problems can develop while making a measurement and the program must be equipped to deal with them. When an error is encountered by the instrument, it stores an error code and description in a special buffer which can be retrieved by the program (usually using the “:SYST:ERR?” command in SCPI compliant instruments) which can in turn be used in the program’s error handling routines. SCPI (Standard Command for Programmable Instruments) is a series of industry-wide instrumentation standards regarding how instrument commands are structured

and contains several common commands. Another advantage of this strategy is that the programmer can define their own errors even if there are no real programmatic or link errors. For example, the programmer can set limits on the measurements such that if the measurement moves out of a certain range, for instance the signal-to-noise ratio gets too low, the program will stop and an error message will appear. With well-designed and implemented error handling, this is quite easy.

After the measurements have been made, the program can make any necessary calculations such as averaging, minimum and maximum values, or any number of other statistical calculations. Two related reasons exist for doing these calculations here rather than after all of the data has been gathered. First is

memory management. This is not really a problem when the data set is ten points, but if the data set has ten million points, it can cause a slowdown in performance. Second, it can be difficult to run calculations on large data sets. Finding the average of ten million points, for example, could take as long as a few seconds, depending on the computer, the algorithm, and the language in which the program is written. On the other hand, if the data is calculated as the data comes in, there is no delay at the end of the program and far less memory is required. Less memory is needed because the new data can be written over the old data in memory as it comes in since the old data has already been stored in the output file. Additionally, even if the calculation is quite complex, the performance slowdown can be worked into the delay function of the program and, as such, there will not be any noticeable delay. A ten-second calculation makes little difference to the user if it is completed during a ten minute delay between measurements.

The next step is to write the data to the text file. In most cases, this involves only converting the data to a string format and then sending that string to the output file. Most instruments give the option of returning the measurement values in an ASCII format which means the conversion may be unnecessary, the strings need only be concatenated. Additionally, the program must add some recognizable delimiter character between the measurements. This is required so that the database or spreadsheet can

tell when one number ends and the next begins. As mentioned earlier, the most commonly used delimiting characters are *tab* and *comma*. Two complications can force a change in this step. First, if for instance, the measurements produce a table of data it may require that a new text file is created with each iteration. An example of such a case is if the user might want to record channel number, wavelength, power, and signal-to-noise ratio for each channel in a DWDM system. A new text file would have to be set up each time this set of measurements was taken. The second variation can come if it is necessary to output all of the data at one time. An example may be that some calculation must be delayed until all of the data has been gathered, or the output to the text file is too complicated or time consuming so it should be done only once. In such a case, the data would be collected from all of the measurements, whether it is into an array or a string, and then written to the output file only after the program had completed. As each measurement was taken, each individual datum would be appended to the data which had already been collected. Remember, however, that this is done at the cost of memory usage and system performance.

The final step in the loop is for the program to wait until it is time for the next measurement. This step is really the heart of the program. On the surface, it seems very simple, but it is deceptively complex. First of all, the *wait* function must allow the

user to abort the program. The reason for this is if there is some problem and the user needed to stop the program, it would be quite frustrating to wait for the entire test duration to fix a simple problem. Second, the program should use a minimum of system resources during this step. It makes no sense to tie up the computer while this program is simply waiting until the next measurement. At first look, the programmer may want to use a *sleep* or *wait* command for the duration of the measurement period. This function, in effect, stalls the program for a certain duration of time and uses very little system resources.

Unfortunately, this simple solution has a couple of problems. First is that most *sleep* and *wait* functions do not allow the user to abort the function. That is, if the user presses the *abort* button or key, the program will not even look at the *abort* button until the *sleep* function is complete. For example, if the measurements are being taken once every hour, the *abort* command may take an hour to be read, which makes the *abort* button all but useless. It seems that an easy solution to this problem would be to remove the *sleep* function and start a sub-loop that repeatedly checked if the user entered the *abort* command until the specified time limit was reached. The problem with this approach is that it uses far too much processor power as the program is occupying system resources to perform these tasks, but is not doing any useful work. The optimum solution lies somewhere in the middle between using a single *sleep* function for

the duration of the delay and using a loop without any *sleep* function. That is, for the program to sleep for a short period of time, check for the *abort* command, and then loop until the *abort* command is seen or the time limit is reached. The sleep time could be a half-second, fast enough that the user may not even notice the delay if they abort the program, but far long enough that the processor can perform many other tasks in the mean time. The overall wait function would then look like something like the code in Figure 3.

The second complexity is that the measurements and even sometime the writing of the file can take seconds or in some cases even minutes and may not require exactly the same amount of time with each iteration. As such, it is necessary to subtract the completion time from the other steps in the loop from the overall wait time so that the entire process takes a consistent

and predictable amount of time. Measuring this completion time is usually easy enough to accomplish by simply storing the system time when the measurement and file writing processes start and end and then finding the difference of the two. The required cycle wait time is then the total wait time minus the measurement time.

At this point, it is time to repeat the entire measurement loop. The loop should be repeated until an error is encountered, an abort command is given, or the time limit is reached. If properly coded, the abort command can be treated as an error. By doing this the loop need only check for an error or the time limit instead of checking for an abort command separately.

The error-handling function should be performed after the program exits the loop. All this function needs to do is output a description of the error encountered. The error can simply be

coded as a number; for instance error number one is a failure to open the file, error number two is a GPIB problem, and so on. The error catching function would consist basically of a lookup table that would take the error code as a number and output the description.

One can completely automate the process of both the measurement and data transfer and storage processes using simple remote programming. This means that no operator is needed to press the buttons on the instrument nor write down the results and type them into a database, it is all done in the program. The program required for such a process is not overly complex nor is it by any means extremely simple.

```
//wait time - the amount of time to wait (in milliseconds)
//stop - boolean whether the user has aborted the program

j = 0; //initialize a counter
while (j < wait_time && !stop){ //loop until time limit or
  abort
    sleep(500); //sleep .5 seconds
    stop = user_abort(); //check for user abort
    j = j+500; //increment the counter
}
```

Figure 3. Pseudo code example of wait function

Appendix A: Visual C++ Example

The following is an example of a datalogging program written in Microsoft Visual C++. It is important to note that no actual measurements are taken in this program in the interests of generality. Instead, dummy functions which return random numbers are used as place holders.

```
/*-----
Program:      Datalogger v1.0
Start Date:   August 25, 2000
Last Modified: September 7,2000
Language:     Visual C++, v6.0
Description:  Datalogger is an example program which demonstrates how a test can repeated at a
              specified interval over a period of time. The data is then logged into a tab delimited stan-
              dard ASCII text file which can be imported into a spreadsheet or database program.
              To make the program more general, no actual measurements are taken. Instead, dummy
              methods are used to return dummy values to illustrate the concepts, rather than actually make
              a measurement.
```

The basic program structure is as follows:

1. Set up the instrument and the output file(s)
2. Perform measurement and gather data from the instrument
3. Send the data to the output file and perform running calculations (e.g. averaging, min, max)
4. wait specified time (e.g. 1 minute)
5. repeat 2-4 until the time limit or asked to stop

Note: This example is provided as an illustration "as is", and Agilent Technologies makes no warranty of any kind with regard to this example.

```
-----*/
//LIBRARY CALLS
#include <windows.h> //windows functions
#include <stdio.h> //standard i/o library
#include <conio.h> //more i/o functions
#include <time.h> //time functions
#include <direct.h> //directory functions
#include <io.h> //file io functions
#include <sys/stat.h> //constants
#include <stdlib.h> //standard functions
#include <fstream.h> //File i/o library

//PROTOTYPES
int wait(int tme, time_t start);
int creatFile(char path[], char fileName[]);
int Measures(double *ans, time_t *tmStmp);
int Msrmnt1(double *ans);
int Msrmnt2(double *ans);
int Msrmnt3(double *ans);
int DataToString(double data[3], time_t TmStmp, char *dat);
int Get_Time(char name[], long *num);
int Get_String(char name[], char *str);
int Error_Catch(int Error_Code);
int Inst_Setup(int Addr);
int init_stats(double measures[3], double *max, double *min, double *mean);
int stats(double measures[3], double *max, double *min, double *mean, int j);
int print_stats(double max[3], double min[3], double mean[3]);

/*-----
```

Function: Main (INCOMPLETE)

Description: The main function is the backbone of the program. It makes the calls the various other methods and functions and controls program flow.

Inputs: None

Output: int - represents an error, if any.

```

_____/
int main( void ){
//Declarations
    time_t start, stamp;           //the start time of the program and timestamp holder
    double meas[3], *mes;          //an array for holding measurement data and a pointer
    double max[3], min[3], mean[3]; //arrays for holding min/max/mean
    double *mx, *mn, *ave;         //pointers to those arrays
    int i = 0, err=0;              //int counter and error code
    long duration, period;         //long ints for test duraion and period
    char data[256], *dat;          //a string for the data and a pointer to it
    char file[256], *fil;         //a string for the file name, and a pointer
    char path[256], *pat;         //a string for the path, and a pointer

    mes = meas;                   //assign the pointers to the arrays
    dat = data;
    pat = path;
    fil = file;
    mx = max;
    mn = min;
    ave = mean;

    err = Get_String("File Name (e.g. c:\\loggeddata\\)", pat); //prompt for the path
    if(err !=0) return Error_Catch(err); //check for errors

    err = Get_String("Path (remember the file extension '.txt')", fil); //prompt for the file name
    if(err !=0) return Error_Catch(err); //check for errors

    printf("data will be written to %s%s.\n",path, file);

//print the path and file name

    err = creatFile(path, file); //create output file
    if(err !=0) return Error_Catch(err); //check for errors

    ofstream out(file); //declare output stream "out"
    if (!out) return Error_Catch(-1); //exit if an error occurs while opening the file

    err = Get_Time("duration", &duration); //prompt the user for test time
    if(err !=0) return Error_Catch(err); //check for errors

    err = Get_Time("period", &period); //prompt user for the measurement period
    if(err !=0) return Error_Catch(err); //check for errors

    printf("duration:\t%d\tperiod:\t%d (secs)\n", duration, period);
    //print the test duration and period in seconds

    err = Inst_Setup(717);
    if(err !=0) return Error_Catch(err); //check for errors

    time (&start); //find the start time of the program; the current time

    err = Measures(mes, &stamp); //take the first Measurements

```

```

if(err == 0){
    DataToString(mes, stamp, dat);
    printf("%s", data);
    out <<data;
    init_stats(meas, mx, mn, ave);
} //if(err == 0)

while(i*period<duration && (err == 0) && ((err = wait(period, start))==0)){
    //loop until the test duration is completed or the program is aborted
    err = Measures(mes, &stamp);
    if(err == 0){
        DataToString(mes, stamp, dat);
        printf("%s", data);
    out <<data;
    time (&start);
    i++;
    stats(meas, mx, mn, ave, i);
} //if(err == 0)
} //for()

print_stats(max, min, mean);
return Error_Catch(err);
} //main()
/*_____

```

Function: Wait

Description: The WAIT function simply stalls the computer between measurements. It polls the keyboard to see if any key was pressed, if so, the program stops and an error message is displayed. This is to allow the user to abort the program at any point. The method uses the sleep() method (in windows.h) to save on processor usage during the stalls.

Input: int time - specifies the number of seconds for which the method is to run.

Output: int - represents an error, if any.

0 - no error

1 - program aborted

```

/*_____*/
int wait(int tme, time_t start){

    //Declarations
    int ch1;
    time_t finish;
    double elapse;

    //ch1 is used to hold keyboard data, i tracks the # of runs
    //finish is the finish time of the method
    //elapse is the run time of the method

    do
    {
        Sleep(500);
        ch1 = _kbhit();
        elapse = difftime(time (&finish), start);
    } while((ch1 == 0) && (elapse<tme));

    //repeat until time expires or a key press
    if (elapse < tme){
        getch();
        return 1;
    } //if
    else{
        return 0;
    } //else

} //wait()
/*_____

```

Function: creatFile()

Description: the function creatFile creates a text file with a given name in a given directory. If and Error is encountered, an error message appears on the debug screen

Inputs:

path[] string (char[]) naming the destination directory

filename[] string (char[]) naming the destination file

Outputs:

int -1 error in creating the file or changing directory

0 no error

-----*/

```
int creatFile(char path[], char fileName[]){
```

```
int dr, dr1, fil;           //variable declarations used for error checking
                           //dr stores the output of chdir()
                           //dr1 is used to recheck if the first chdir() fails
                           //fil stores the output of creat()
```

```
dr = chdir(path);          //change directory to the designated path
```

```
if (dr!=0){                //check for errors in changing directory
    dr1 = chdir("c:\\");    //change the directory to 'c:\'
    if (dr1 != 0) return -1; //if that fails, return error
    dr1 = mkdir(path);     //try to create a new directory
    dr1 = dr1 + chdir(path); //try to move to that directory
    if(dr1!=0)             //recheck for error
        return -1;        //return error code and quit function
} // if(dr != 0)
```

```
fil = _creat(fileName, S_IWRITE); //creates a file named as designated
```

```
if (fil == -1)             //check for errors in creating the file
    return -1;            //return error code and quit function
```

```
return 0;
```

```
} //creatFile
```

/*-----*/

Function: Measures()

Description: the Measures function calls the individual measurement functions and adds a time stamp and then returns the data as a "Data" structure.

inputs: *ans[] - a pointer to the destination of the measurement data

*tmStp - a pointer to the destination of the time stamp

outputs: int - error code

0 - no error

2 - measurement error

-----*/

```
int Measures(double *ans, time_t *tmStmp){
```

```
//Declarations
```

```
int err=0;           //error code
```

```
err = Msrmnt1(ans);  //take the first measurement
```

```
if (err == 0)        //check for error
```

```

    err = Msrmnt2(ans+1);    //take the second measurement
if (err == 0)               //check for error
    err = Msrmnt3(ans+2);  //take the third measurement

time(tmStmp);              //current time is the timestamp

return err;
} //Measures()
/*
Function: Msrmnt1()
Description: Msrmnt1 simply a dummy function which returns a random double. It should be
replaced by a function which actually takes a measurement.

inputs: double *ans - pointer to the answer destination
output: int - error catcher
         0 - no error
         2 - measurement error
-----*/
int Msrmnt1(double *ans){
double num = rand();        //create a random number
*ans = num;                 //place it in the pointed location
return 0;                   //return "no error"
}
/*
Function: Msrmnt2()
Description: Msrmnt2 simply a dummy function which returns a random double. It should be
replaced by a function which actually takes a measurement.

inputs: double *ans - pointer to the answer destination
output: int - error catcher
         0 - no error
         2 - measurement error
-----*/
int Msrmnt2(double *ans){
double num = rand();        //create a random number
*ans = num;                 //place it in the pointed location
return 0;                   //return "no error"
}
/*
Function: Msrmnt3()
Description: Msrmnt3 simply a dummy function which returns a random double. It should be
replaced by a function which actually takes a measurement.

inputs: double *ans - pointer to the answer destination
output: int - error catcher
         0 - no error
         2 - measurement error
-----*/
int Msrmnt3(double *ans){
double num = rand();        //create a random number
*ans = num;                 //place it in the pointed location
return 0;                   //return "no error"
}
/*

```

Function: DataToString

Description: DataToString converts the data (an array of three doubles) and the time stamp into a string, which can then be exported to the text file.

inputs: double[3] data - the measurement data
 time_t TmStmp - the time stamp
 char *dat - pointer to the soutput string

Output: int - error code (always 0)

```

_____*/
int DataToString(double data[3], time_t TmStmp, char *dat){

struct tm *StructTime;          // a time structure which will hold the time stamp

StructTime = localtime(&TmStmp); //convert the time_t to a tm structure

sprintf(dat, "%f\t%f\t%f\t%s",data[0],data[1],data[2], asctime(StructTime));
    //build the string and store it in the pointer location

return 0;
} //dataToString()

```

/*_____*/

Function: Get_Time()

Description: this function prompts the user for a time value in hh:mm:ss format and then returns the number of seconds.

inputs: Char[] name - the name of the time to be propted
 long *num - a pointer to the total number of seconds

output: int - error code
 0 - no error
 3 - user input error

_____*/

```

int Get_Time(char name[], long *num){
    int hours, minutes, seconds;          //declarations

    if(sprintf("What is the test %s (hh:mm:ss)?", name) <0) //prompt the user
    return 3;                               //and check for error

    if(sprintf("%d:%d:%d", &hours, &minutes, &seconds) != 3) //scan the input
    return 3;                               //and check for error

    *num = hours*3600 + minutes*60 + seconds;//calculate the total number of secs

    return 0;
} //return "no error"
}
/*_____*/

```

Function: Get_String()

Description: this function prompts the user for a string value.

inputs: Char[] name - the name of the string to be propted
 long *str - a pointer to the total number of seconds

output: int - error code
 0 - no error
 3 - input error

_____*/

```

int Get_String(char name[], char *str){

    if(sprintf("What is the %s?", name) <0) //prompt and check for error
    return 3;

    if(scanf("%s", str) != 1) //scan the user input and check for error
    return 3;

    return 0; //return "no error"
}
/*_____*/

```

Function: Error_Catch()

Description: Error catch is a very simple error handling function which prints the error to the output screen then echos the input

Possible Error Codes:

- 0 - no Error
- 1 - Error in opening or creating the output file
- 1 - Program Aborted by the User
- 2 - Error in Measurement
- 3 - Error in User input
- 4 - Instrument setup error

input: int errorCode - the code of the error present

Output: int - error code (identical to the input)

_____*/

```

int Error_Catch(int Error_Code){
switch(Error_Code){
    case 0: //no error case
        printf("Program Completed Sucessfully!\n");
        break;
    case -1: //file error case
        printf("ERROR: Cannot create/open output file\n");
        break;
    case 1: //program aborted
        printf("ERROR: Program aborted by the User\n");
        break;
    case 2: //measurement error
        printf("ERROR: Error in taking measurement\n");
        break;
    case 3: //User input error
        printf("ERROR: User input Error\n");
        break;
    case 4: //instrument setup error
        printf("ERROR: Cannnot setup instrument interface\n");
} //switch(Error_Code)
return Error_Code;
} //Error_Catch()
/*_____*/

```

Function: Inst_Setup()

Description: this is a very simple dummy program which simply returns 0. In an actual test system, this would be replaced by code to set up the instrument, now it is simply a place holder.

inputs: int Addr - address of the instrument

output: error code 0 - no error

4 - instrument set-up error

-----*/

```
int Inst_Setup(int Addr){
```

```
//insert instrument setup code here
```

```
return 0;
```

```
}//Inst_Setup()
```

-----*/

Function: init_stats()

Description: init stats simply copies the first measurement's data into the min, max and mean arrays.

inputs: double[3] measures - the measurement data

double *max - pointer to the maximum arrays

double *min - pointer to the minimum array

double *mean - pointer to average array

output: int - error code (always 0)

-----*/

```
int init_stats(double measures[3], double *max, double *min, double *mean){
```

```
for(int i=0; i<3; i++){ //cycle through measurements
```

```
    *max = measures[i]; //place the data in the max....
```

```
    *min = measures[i]; //min.....
```

```
    *mean = measures[i]; //and mean arrays.
```

```
    max++; //increment the pointers before moving to the
```

```
    min++; //next measurement
```

```
    mean++;
```

```
}//for
```

```
    return 0;
```

```
}//init_sata()
```

-----*/

Function: stats()

Description: stats updates the running stats (min, max, and mean). It is very similar to init_stats.

inputs: double[3] measures - the measurement data

double *max - pointer to the maximum arrays

double *min - pointer to the minimum array

double *mean - pointer to average array

int j - the number of measurements, which have been taken

output: int - error code (always 0)

-----*/

```

int stats(double measures[3], double *max, double *min, double *mean, int j){
for(int i=0; i<3; i++){ //cycle through measurements

    if (measures[i] > *max)
        *max = measures[i]; //place the data in the max....

    if (measures[i] < *min) //min.....
        *min = measures[i];

    *mean = (measures[i] + (*mean) * j)/(j+1); //and mean arrays.

    max++; //increment the pointers before moving to the
    min++; //next measurement
    mean++;
} //for
return 0;
} //init_sata()
/*_____

```

Function: print_stats()

Description: print_stats simply prints the running stats at the end of the program.

inputs:

double[3] max - the maximums of the measurements

double[3] min - the minimums of the measurements

double[3] mean - the arithmetic means of the measurements

outputs:

int - error code (always 0)

```

_____*/
int print_stats(double max[3], double min[3], double mean[3]){
printf("max value:\n%f\t%f\t%f\n", max[0], max[1], max[2]);
printf("min value:\n%f\t%f\t%f\n", min[0], min[1], min[2]);
printf("ave value:\n%f\t%f\t%f\n", mean[0], mean[1], mean[2]);
return 0;
} //print_stats()

```

Appendix B: LabView Example

The following figures show a National Instruments LabView version 5.1 example of a datalogging program. The first figure is a program hierarchy illustration followed by pictures of each individual program function.

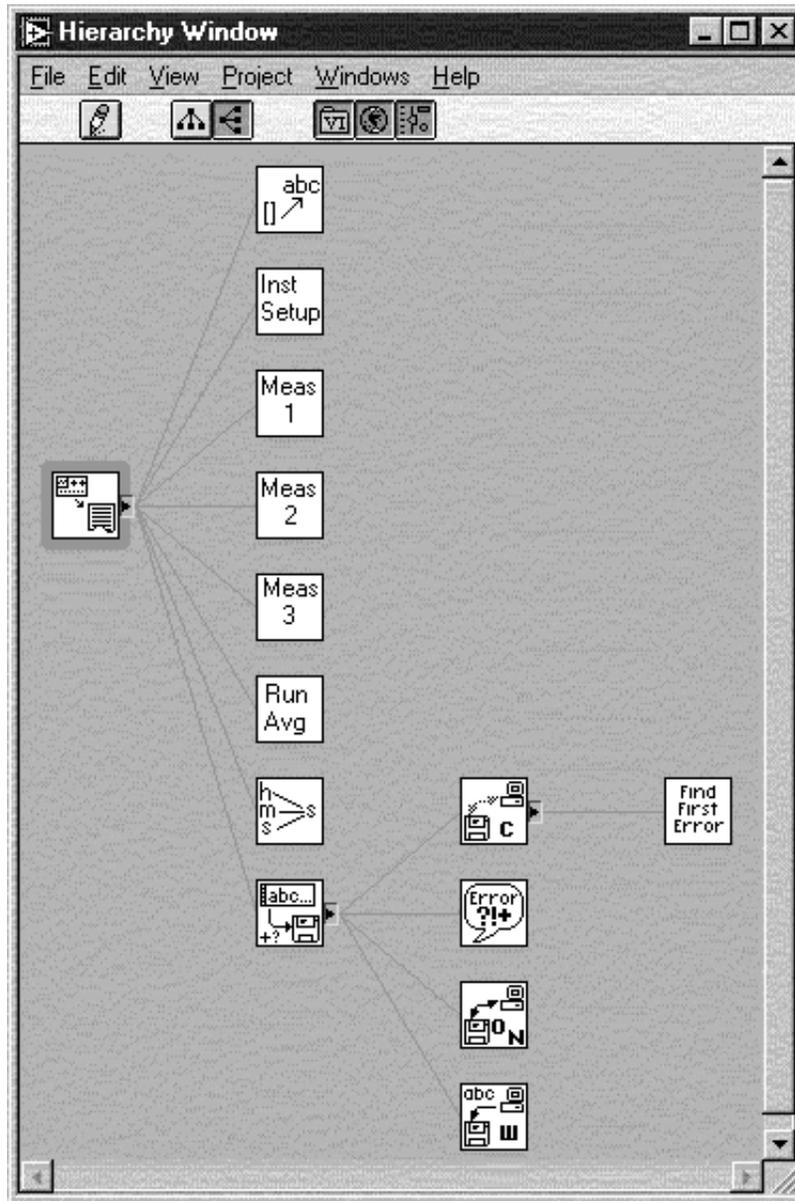


Figure B-1. Hierarchy of the LabView datalogging example.

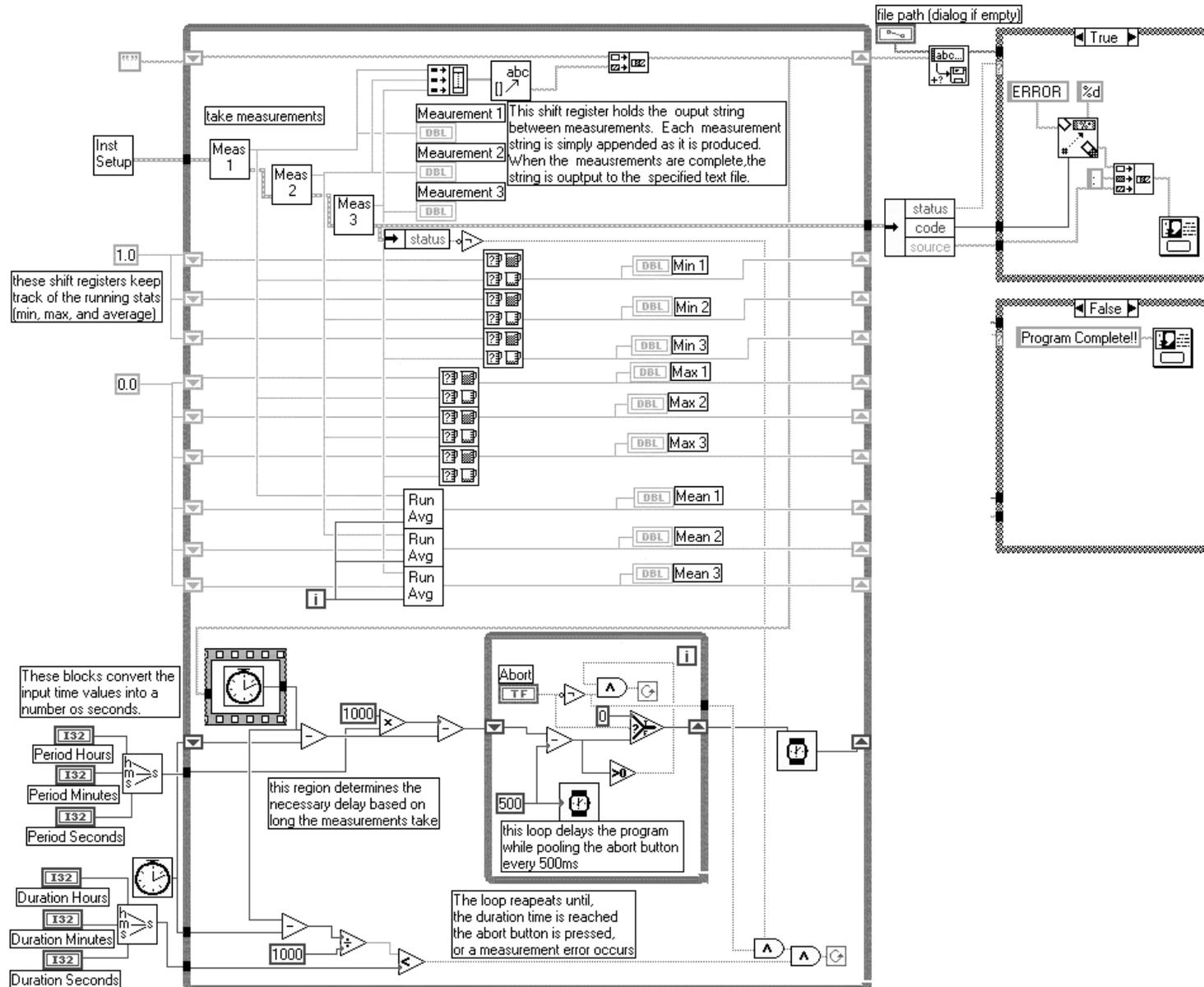


Figure B-2. The main program module of the LabView datalogging program.

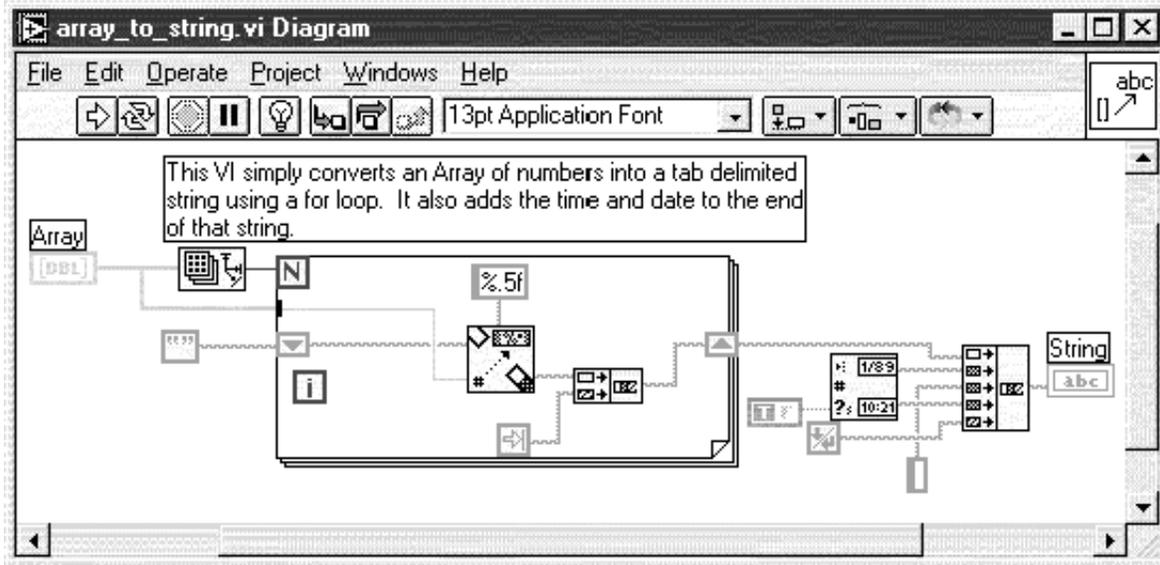


Figure B-3. The *array_to_string* function converts the raw data into a string with a time stamp which can then be written to the output file.

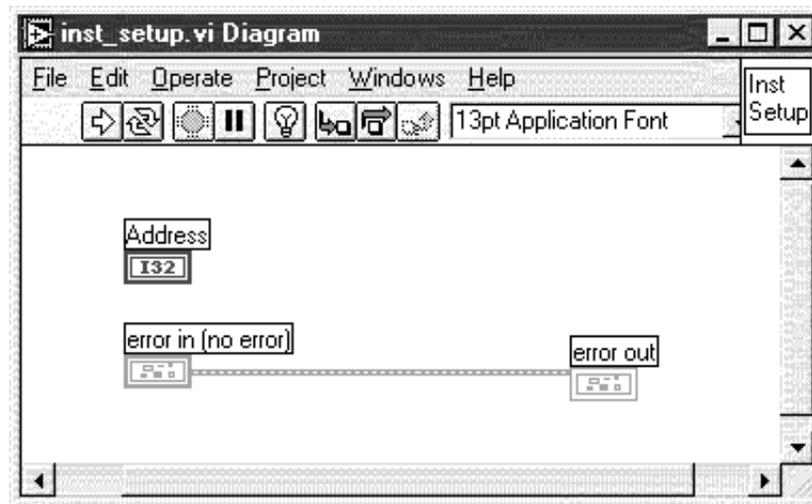


Figure B-4. The *inst_setup* function is a dummy function. It's only functionality is to pass on any errors passed to it. This is where the instrument setup functionality should be inserted.

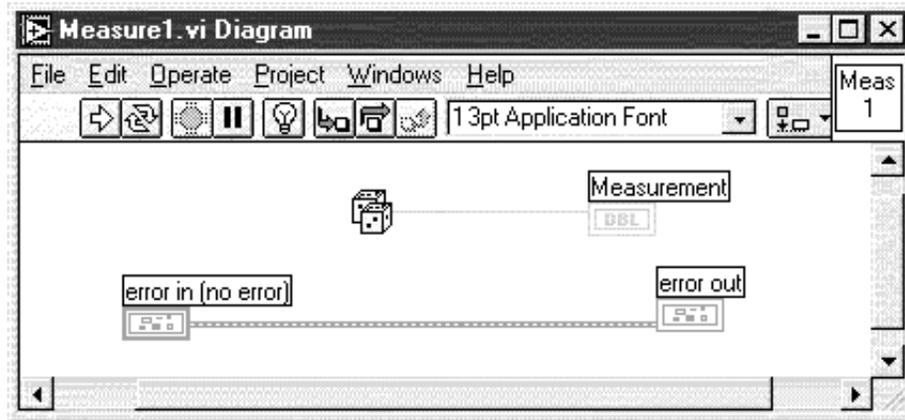


Figure B-5. The measurement functions are also dummy functions. They simply return a random number. Shown is the first of three identical functions. These functions would be replaced with actual measurement functions.

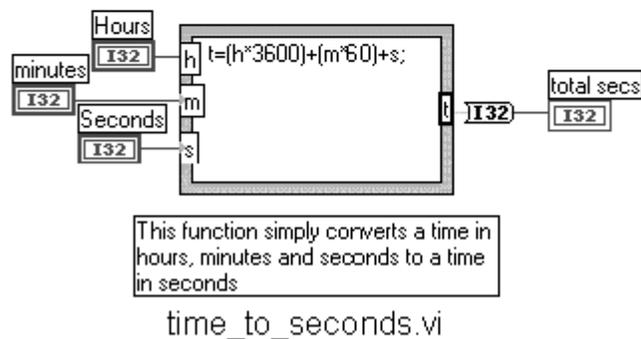
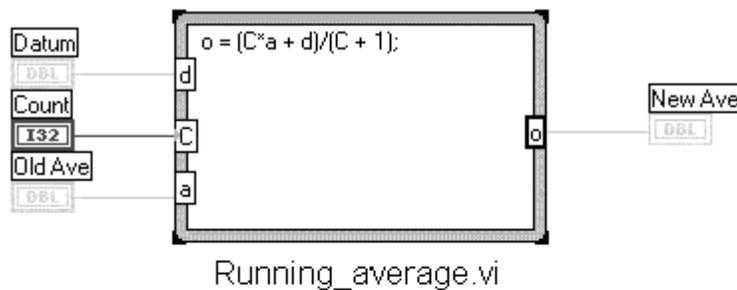


Figure B-6. The *running_average* and *time_to_seconds* functions are simple mathematical functions. They are put in separate files in order to save space in the main function.

All of the other functions are standard LabView v5.1 modules. See the LabView help files for documentation on these modules.

Agilent Technologies'

Test and Measurement Support, Services, and Assistance

Agilent Technologies aims to maximize the value you receive, while minimizing your risk and problems. We strive to ensure that you get the test and measurement capabilities you paid for and obtain the support you need. Our extensive support resources and services can help you choose the right Agilent products for your applications and apply them successfully. Every instrument and system we sell has a global warranty. Support is available for at least five years beyond the production life of the product. Two concepts underlie Agilent's overall support policy: "Our Promise" and "Your Advantage."

Our Promise

Our Promise means your Agilent test and measurement equipment will meet its advertised performance and functionality. When you are choosing new equipment, we will help you with product information, including realistic performance specifications and practical recommendations from experienced test engineers. When you use Agilent equipment, we can verify that it works properly, help with product operation, and provide basic measurement assistance for the use of specified capabilities, at no extra cost upon request. Many self-help tools are available.

Your Advantage

Your Advantage means that Agilent offers a wide range of additional expert test and measurement services, which you can purchase according to your unique technical and business needs. Solve problems efficiently and gain a competitive edge by contracting with us for calibration, extra-cost upgrades, out-of-warranty repairs, and on-site education and training, as well as design, system integration, project management, and other professional engineering services. Experienced Agilent engineers and technicians worldwide can help you maximize your productivity, optimize the return on investment of your Agilent instruments and systems, and obtain dependable measurement accuracy for the life of those products.

By internet, phone, or fax, get assistance with all your test & measurement needs.

Online assistance:

www.agilent.com/comms/lightwave

Phone or Fax

United States:

(tel) 1 800 452 4844

Canada:

(tel) 1 877 894 4414

(fax) (905) 282 6495

Europe:

(tel) (31 20) 547 2323

(fax) (31 20) 547 2390

Japan:

(tel) (81) 426 56 7832

(fax) (81) 426 56 7840

Latin America:

(tel) (305) 269 7500

(fax) (305) 269 7599

Australia:

(tel) 1 800 629 485

(fax) (61 3) 9210 5947

New Zealand:

(tel) 0 800 738 378

(fax) 64 4 495 8950

Asia Pacific:

(tel) (852) 3197 7777

(fax) (852) 2506 9284

Product specifications and descriptions in this document subject to change without notice.

Copyright © 2000 Agilent Technologies

Printed in USA December 15, 2000

5988-1329EN

UNIX is a registered trademark of The Open Group.

Oracle is a registered US of Oracle Corporation, Redwood City, California.

Microsoft and MS Windows are trademarks of Microsoft Corporation.

Lotus and 1-2-3 are US registered trademarks of Lotus Development Corporation.



Agilent Technologies

Innovating the HP Way