Computer room temperature alarm: design and implementation for UPS

O'Rielly, Barbara

Publisher's version / Version de l'éditeur:
https://doi.org/10.4224/8895984
Laboratory Memorandum; no. LM-2004-23, 2004

National Research Council Canada
Conseil national de recherches Canada
Institute for Ocean Technology
Institut des technologies océaniques

Laboratory Memorandum

LM-2004-23

Computer Room Temperature Alarm: Design and Implementation for UPS

B. O'Rielly

August 2004

Canada

DOCUMENTATION PAGE

REPORT NUMBER	NRC REPORT NUMBER	DATE
LM-2004-23		August 2004
REPORT SECURITY CLASSIFICATION		**DISTRIBUTION**
Unclassified		Unlimited

TITLE

COMPUTER ROOM TEMPERATURE ALARM: DESIGN AND IMPLEMENTATION FOR UPS

AUTHOR(S)

Barbara O'Rielly

CORPORATE AUTHOR(S)/PERFORMING AGENCY(S)

Institute for Ocean Technology, National Research Council, St. John's, NL

PUBLICATION

SPONSORING AGENCY(S)

Institute for Ocean Technology, National Research Council, St. John's, NL

IMD PROJECT NUMBER	NRC FILE NUMBER		
KEY WORDS	**PAGES**	**FIGS.**	**TABLES**
UPS, Computer Temperature Alarm	13, App. A-E		

SUMMARY

During my 4-month employment period with IOT, there have been a considerable amount of problems with the present air conditioning system that is implemented within the computer room. This presents two major problems since: a) The computer room is not accessible to general employees at IOT, and as such, is not constantly monitored; b) The room contains hundreds of thousands of dollars in equipment, and invaluable amounts of information; CPU overheating could cause irreparable damage to this sensitive equipment and lose or corrupt file systems forever.

The purpose of this project was to create and implement a system that would both monitor the status of the ambient temperature of the computer room, and notify the correct personnel when the temperature reaches an undesirable level. This was achieved through the use of the C programming language and the creation of functions, loops, and system calls within the existing *checkups.c* source code.

The UPS is monitored by a program called checkUPS® which polls the UPS status through a buffer string every five seconds. I took advantage of this pre-designed polling system by pulling select values from the string buffer, which updated as the checkUPS® software polled.

The design and implementation of this particular program was successful, as well as time and cost effective. The program is currently in use at IOT for their UPS1 system. There are plans in the near future to implement this system for the UPS2 system and monitor more aspects of the UPS other than the ambient temperature.

ADDRESS National Research Council
Institute for Ocean Technology
Arctic Avenue, P. O. Box 12093
St. John's, NL A1B 3T5
Tel.: (709) 772-5185, Fax: (709) 772-2462

National Research Council Canada
Institute for Ocean Technology

Conseil national de recherches Canada
Institut des technologies océaniques

Computer Room Temperature Alarm: Design and Implementation for UPS

LM-2004-23

Barbara O'Rielly

August 20th, 2004

Title:
Computer Room Temperature Alarm: Design and Implementation for UPS

Prepared for:
Mr. Paul Thorburn, NRC
Mr. John Hudson, Co-op Education

Originating Organization:
Institute for Ocean Technology, NRC

Author:
Barbara O'Rielly

Date:
August 20th, 2004

Table of Contents:

1.0 Introduction

C is the programming language that I chose to write the program for the computer room temperature alarm. C is a general-purpose programming language that features economy of expression, modern control flow and data structures, and a rich set of operators. It has been closely associated with the UNIX system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a "system programming language" because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains. C deals with the same sort of objects that most computers do, namely, characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

The reason I chose C was because the original source code for the CheckUPS® software was written with a C compiler (see Appendix A). CheckUPS® is a power monitoring software package that allows UPS monitoring in OpenVMS, Windows, and various UNIX environments. This software constantly monitors the status of the battery, CPU temperature, ambient temperature, and AC power and alerts users of irregularities in operation, or shutdowns due to malfunction through several different alarms. In the event of a power failure, this software also has the capability to shutdown servers if available runtime becomes too low before power is restored. In addition to these features, it also creates logfiles of events for viewing by the user.

Lastly, to be able to monitor the computer temperature at all times quickly and easily, I used a program called The Multi Router Traffic Grapher (MRTG). MRTG is a tool to

monitor the traffic load on network-links. This program generates HTML pages containing graphical images, which provide a live visual representation of this traffic. MRTG is based on Perl and C and works under UNIX and Windows NT. MRTG is being successfully used within IOT to monitor the activity of various servers, clusters, as well as fiber and copper connections.

1.1 Purpose

The purpose of this project was to create additional lines of code within the CheckUPS® source code to alert the local computer systems group of high ambient temperature in the computer room. This project was authorized by Mr. Paul Thorburn, P.Eng., the head of the computer systems group at IOT. Currently, an alarm exists within the code, however when triggered can only be recognized when within the computer room. The proposed "new" alarm would send an e-mail message to those in the computer systems group alerting them of the current temperature status, therefore ensuring that the proper people are contacted immediately to remedy the problem quickly and efficiently. This will be an important attribute to the existing software code since the air conditioner in the room has been found to be unreliable at times, and hundreds of thousands of dollars worth of electronic and technical equipment is contained within this specific room.

1.2 Scope

The only limitation that was imposed on this project was time, since I was employed with the Institute for Ocean Technology for a mere four-month term. The reading of extensive amounts of documentation and source code began in the month of June and the final lines of code were finished on Friday, August 6, 2004.

2.0 Design and Implementation of Alarm

2.1 Research

The first step that must be taken whenever initiating a project of any scale is to perform some basic research into the subjects involved. For this particular project, there were many aspects that needed to be explored before any groundwork could be performed.

To begin, I requested documentation for both the CheckUPS® software and UPS system. The CheckUPS® software documentation reviewed such topics as 'installing CheckUPS,' 'connecting to UPS,' 'CheckUPS messages,' and 'CheckUPS Files and Procedures,' as well as providing an excellent reference for troubleshooting common problems. This allowed me to familiarize myself with the operations of the software, and specifically, how it reads the UPS status. The UPS user manual generated important information on the make and model of our particular UPS system, as well as normality's and default settings for certain alarm triggers.

In addition to becoming familiar with the software, I also had to educate myself in the C programming language since I had never written any code in C before. Prior to this work term, I had taken a course in C++ programming, which proved to be a valuable resource when decoding C. To aid in my struggle to learn C, I requested some texts from the CISTI (Canada Institute for Science and Technical Information) library, located within IOT. I received three different books from the library and immediately began to

read them. In addition to this, I was also offered supplementary texts from co-workers in the computer systems group that proved very helpful.

This extensive research took three to four weeks, since I had other ongoing projects during the same time period as this one.

2.2 Reading the Code

The next step in the development stage of the alarm was to read the existing source code. This program required the review of 3157 total lines, of which 2187 were actual code. Within this code I identified several lines within the main loop, as well as function definitions, that would be very useful in the writing of my own piece of code (see Appendix A). Once these important portions of the code were identified, I copied them to a text file (*temp alarm.txt*) for quick reference and closer scrutiny. I then found places within the lines of code that would be optimal for the positioning of my own code. Since I want to poll the UPS system at regular intervals of time, the best place to insert my ambient temperature monitor would be within the main loop, which polls the UPS every five seconds. I also concluded that to retrieve specific information from the UPS I would have to use a string buffer, and the best way to achieve this would be by creating a separate function definition for retrieving the ambient temperature.

2.3 Reading the Buffer String

In order to import the ambient temperature from the UPS, I first had to uncover the position(s) the data occupied in the buffer string. To do this I polled the UPS to display the buffer string and read the temperature from the control panel. The temperature on the control panel display read to be 19 degrees Celsius. By analyzing the string, I realized that there were two places within the string that displayed this value. To determine which

position would be used for polling for ambient temperature, I altered the temperature in the computer room by opening all the doors, thereby allowing warmer air to circulate the room and raising the temperature. Once the temperature increased, I watched the buffer string output to see which value had been altered. The two positions that displayed the ambient temperature were position 64 (the tens position) and 65 (the ones position). Once this was done I could call the appropriate *FstrBuff[#]* command.

2.4 Writing the Code

Now that all the groundwork had been laid and the initial obstacles overcome, it was time to start writing the code. For the rest of the project I was paired with co-worker Doug Walsh who is versed in C programming and has a degree in CS from Memorial University of Newfoundland. The first step was to identify a 'game plan' by writing some pseudo code (the "plain english" explanation of the code). Once this was done we could analyze the specifics of the code we wished to write. To begin we started to write a function that would remove the polling data from the string buffer. We created a new set of function definitions, and declarations and paired these with a function call located within the main loop. The first function that was written, *GetAmbTemp* (see Appendix C, i), was written to actually pull the ambient temperature from the string buffer and convert it to a manageable integer value. Our plan of attack was to read the two values from the string located at positions 64 and 65 (see Section 2.2). We then stored these two values in separate *int* (integer) variables with the names *AmbTempTens* (logically named for the tens position) and *AmbTempOnes* (logically named for the ones position). Once we had these values, we needed to return only one integer value from the function. Through testing (see Section 2.5, i), we discovered that our buffer output was

in ascii, rather than integer values. To remedy this problem we subtracted 48 (see Section 2.5, i) from both string buffer readings, and made our singular return value calculation by multiplying the *AmbTempTens* by 10, and adding *AmbTempOnes*.

The next part that was to be written was the commands that printed the temperature and status every time the checkUPS® polled the UPS system (every 5 seconds). To do this, we located the main *while* (a statement in C programming) loop in the program and inserted our code at the end of the loop. The first step that was taken was to place the output of the function *GetAmbTemp* into a variable (*WhatAmbTemp*), this way we would not have to keep calling the function. After this, we used the command *sprintf* (which prints formatted data to a string), using the buffer variable *errorbuf* (buffer to store the resulting formatted string) and the format "`Temp OK on UPS1`" (string that contains the text to be printed). This took care of the need to print the status of the UPS system. At this stage, the *if* statement did not contain the line which decremented the *MailTimer* variable, this was written at a later date. However, the *if* statement was implemented to create an alarm status string within the loop. When the temperature rises above 23 degrees Celsius, the program will enter the if statement and change the *errorbuf* message from "`Temp OK on UPS1,`" to "`UPS1 Temp Alarm !!!`"

The next line in the main loop allowed for the printing and logging of the UPS temperature. This again utilized the command *sprintf*, except in this case, the buffer variable used was *msgbuf*, and the format was "%d is the UPS1 temp." This piece of code inside the *sprintf* command was then followed by the variable name *WhatAmbTemp*, which stores the value from the function *GetAmbTemp*. The value that is contained

within this variable will then be printed in place of the %d in the format portion of the string because the '%' placeholder contains the following:

```
% [flags][width][.precision][modifiers]type (where type is the most
significant and defines how the value will be printed).
```

Since the letter after the % is d, this signifies that the output will be a signed decimal or integer (see Appendix D).

Also found within the *if* and *else* statements, is a call of the function *TempStamp*. The *TempStamp* function (see Appendix C, iv) was created to enter the temperature from the UPS into a file. The parameters that were entered into this function for manipulation were the directory and name of the logfile (the file that the information will be written into), *errorbuf*, and *msgbuf*, which were temporarily stored within the function in the variables *char *logname , char *logtext1, and char *logtext2* respectively. Within this function, a logfile entry into the file *UPStemp.log* was created containing the following information; a) the time stamp (a.k.a. the date and time the log entry was created); b) the *sprintf* command to print the status string; c) the *sprintf* command to print the temperature in the computer room. It then outputs a dotted line to separate the logfile entries. This part of the code is located within the *if* and the *else* statements so that no matter what the ambient temperature, the program will create a logfile entry.

The final part of writing the code involved getting the program script to actually send an alert to computer systems personnel. This is where the extra lines of code come into play. Before we could write the code containing *mailx* (`mailx` helps you read and send electronic mail messages to specified users) we first had to create an alias within the *mailx* command that contained all the correct addresses for our contact list. To do this we enlisted the help of coworker Gilbert Wong (CS degree from Memorial University). He

added two aliases to the system, one which contained all the addresses for the computer systems employees, as well as text message addresses for their cell phones (alias = checkups), and a second ***test*** alias containing only Doug's @nrc.ca address and cell phone text address (alias = walshd). Once this was completed we discussed the problem concerning the frequency of mailing, since we did not want our alarm to be triggering a *sendmail* command every five seconds (for obvious reasons). We decided upon once an hour after the initial triggered mail message for the program to send out additional warning mail. The trigger for the *mailx* command was through the variable *MailTimer*, which was initialized to a value of 1. When the temperature rises above 23 degrees, the variable is decremented once every 5 seconds until it reaches zero, since the variable is initialized to a value of one, there will be a mail message sent the first time the temperature rises over 23. Also, once *MailTimer* reaches 0, it will be reset to a value of 720 (this represents a period of one hour since one loop completes every 5 seconds). Once this happens, the cycle starts again; it will decrement every 5 seconds until it reaches zero, send a mail message, and reset to 720. This will continue until the temperature drops below 23 degrees.

2.5 Creating a web-based monitoring system

The only problem with our program was that all the data was written into a log file, which was both tedious to view, and not easily accessible. To solve this problem we decided to take advantage of a piece of software that is readily available within the CS group called MRTG (Multi Router Traffic Grapher). To ensure that this program would properly analyze our data, we performed several tests using static data (see Section 2.6, iii). We copied existing MRTG code from another device that was being monitored and

edited this code to suit our specific needs. The highlighted sections of the code in Appendix E, (i), are places in the code where we manipulated the variables and titles to make the analysis of the graph both easier for those viewing, and more logical for those analyzing the data. We then created a piece of code containing the data from the checkUPS® software that is continually updating. To create this code we used another programming language, *Python*, which is an interpreted, interactive, object-oriented programming language similar to Java. We then copied this data into the MRTG directory and allowed it to access this varying data, updating once every three minutes. Once this was completed all members of the computer systems group could readily view the html graph by simply entering the user name and password at the local MRTG page (see Appendix E).

2.6 Testing

The crux of any good program is testing to ensure that all major errors have been ironed out. During different phases of our coding we conducted tests of the existing lines to ensure that we were receiving correct results and values.

(i) The first test that we conducted occurred early in our program writing. We tested our function, *GetAmbTemp* to see if it printed the correct ambient temperature value. However, when we compiled our program and did a test run, we discovered that we were receiving an error in our ambient temperature value. The displayed value was telling us that the temperature in the computer room was 546 vs. the actual temperature in the computer room of 18 degrees Celsius. Quickly we came to realize that the problem did not lie in our coding techniques, but in the output of the buffer string. It seems that the values stored in the buffer array were written in *ascii* code (the code that computers use

to represent characters as binary numbers). To fix this problem we initially tried to use a call within the program called *atoi* (ascii to integer). To do this we wrote the following:

```
AmbTempTens = atoi(ups -> FStrBuff[64]);
AmbTempOnes = atoi(ups -> FStrBuff[65]);
```

However, the *atoi* function only works on information that is type *char* (character) and the information that is extracted from the buffer string is of type *int* (integer). This caused an error when the program was compiling. We decided that we would try and force the output from the string buffer to become a char type, however this was not successful and we still received a compiler error.

Example:

```
AmbTempTens = atoi(char(ups -> FStrBuff[64]));
AmbTempOnes = atoi(char(ups -> FStrBuff[65]));
```

Finally, we realized that in ascii code, the difference between the actual number and its binary representation is 48 (see Appendix B). Therefore, if we wrote the code to subtract 48 from the string buffer readings, the ambient temperature calculation would be valid. This was the approach that worked for all numbers, therefore, this was written into the return value for our function (see Appendix C, ii).

(ii) The second test that was conducted was on the mail commands to ensure that they not only sent messages, but also sent them at regular intervals. To do this, we set the temperature trigger to 14 degrees (since this was lower than the temperature in the computer room); this way the program would enter the mail loop. We also adjusted the frequency of the mailer, changing it from every 720 cycles (one hour), to 12 cycles (one minute). We then opened the file *UPStemp.log* and forced it to update every 5 seconds so we could constantly monitor the log entries. As you can see, there are two lines commented out (blue font) within the *MailTimer if* statement. These lines were

uncommented for the test, and the actual mail script that is used became commented. This was done so that we would only receive a printout in the logfile to verify that the loop was executing at regular intervals instead of a myriad of emails. Once this was confirmed, we allowed the mail script to run again and replaced the mail alias from *checkups* to *walshd*. This ensured that the *mailx* script was executing properly without unnecessarily disturbing the entire mailing list. We allowed the loop to run with all these test conditions and once it executed properly, we reset the conditions to meet the original specifications. The script was then forced to execute one more time and it was concluded that the results that were obtained were desirable.

(iii) The third test was of the web-based monitoring system created through the use of the MRTG. To test this system of monitoring, we created a static text file in notepad, and used this as the input for the MRTG. It created a graphical representation of the test value, which was a linear graph of 20 degrees Celsius. Once we knew that the MRTG program was reading from the correct directory and had applied the correct variable labels and titles, we inserted the correct file with the correct code into the MRTG input and watched the html output graph through *Internet Explorer*.

3.0 Conclusion

The computer room temperature alarm has been implemented for the UPS1 system and is functioning as per Mr. Thorburn's specifications. The project was completed on August 6th, 2004, and the monitoring system has been running since August 10th, 2004. Several revisions to the code have been made since the completion date; they do not change the operation of the program, but make the coding 'tighter,' and more compact.

These lines of code operate on startup of the operating system with the existing checkUPS® software. Also, documentation on the operation of our additional code and procedures for execution have been made accessible to the other members of the CS group via network drives.

4.0 Recommendations

There are several recommendations that could be made to improve the efficiency and practicality of this project.

1.) Revision of source code: The existing source code is at times convoluted and confusing. A closer inspection of this code by trained programmers could result in not only 'tighter' lines of code, but also fewer lines of code. This not only makes the program more visually appealing, but also allows for easier troubleshooting should the need arise.

2.) Additional String Buffer Manipulation: This program script does not only have to be used to monitor the status of the ambient temperature of the UPS1 system, it can also be implemented to monitor such things as frequency, number of watts, VA output, and battery status.

3.) Additional UPS monitors: Presently, the temperature alarm and UPS monitoring system only monitors UPS1 in the building. There is currently one other UPS system managed by the CS group at IOT. This system (UPS2) could be set up with the same monitoring capabilities using this code.

References:

1. Cooper, James W.; & Richard, Lam B., *A Jumpstart Course in C++ Programming*, John Wiley & Sons, Inc. (1994)

2. Holub, Allen I., *Enough Rope To Shoot Yourself In The Foot: Rules for C and C++ Programming*, McGraw-Hill (1995)

3. Kerninghan, Brian W., *The C Programming Language, Second Edition*, (1988)

4. Porter, Anthony, *The Best C/C++ Tips Ever*, Osbourne McGraw-Hill (CA,1993)

Appendix A:

The original code

i) How to retrieve data from string buffer:

```
        /* Check for Hi temp alarm */
        if (ups->UpsType != UPSTYPE_FORTII)
        {
                if (ups->CheckTemp) ups->AlarmFlag |= (8 & ToBinary(ups->FStrBuff[20]));
        }

        ups->FStrBuff[62] = '\0';             /* mark end of remaining time */
        GetNum(&ups->FStrBuff[58],&ups->Runtime);

        /* is there an alarm present ? */
        if ( ups->AlarmFlag || ((ups->Runtime <= ups->CutOff) && ups->InverterOn) )
        {
                if ( 2 & ToBinary(ups->FStrBuff[21]) )
                {
                        ups->MsgType = UPSMSG_LOWBAT;
                }
                else
                {
                        ups->MsgType = UPSMSG_ALARM;
                }
                return BP_TRUE;
        }

        /* or just warn everyone about time remaining */
        if ( ups->InverterOn )
        {
                ups->MsgType = UPSMSG_INVERTER;
                return BP_TRUE;
        }

        /* Check if last msg was UPSMSG_COM_LOST i.e. Communication resumed */
        if ( (UPSMSG_COM_LOST == ups->MsgType) && ups->UpsDataString )
        {
                ups->MsgType = UPSMSG_COM_RESTORED;
                return BP_TRUE;
        }

        /* else no message required */
        ups->MsgType = UPSMSG_NO_MESSAGE;
        return BP_FALSE;
break;
```

Appendix B:

Ascii Conversion Table

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	¶	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	|
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂

Appendix C:

The New Code

(i) The Creation of New Variables

```
/*************************************************************************/
/*****************************  MAIN  ************************************/
/*************************************************************************/

#ifdef USE_PROTOTYPES
int main(int argc, char **argv)
#else
main(argc, argv)
int argc;
char **argv;
#endif
{
    int i,j;
    int WhatAmbTemp;
    int TempAlarm;
    int upsmodel;
    int varate;
    char *p;
    char errorbuf[512];
    char msgbuf[512];
    char LogString[512];
    char cmdbuf[256];
```

(ii) The GetAmbTemp( ) Function

```
/*************************************************************************
* Function Name: GetAmbTemp()
*
* Author(s): Doug and Barb
*
* Description: Gets the Amb Temp
*
* Entry Values: none
*
* Exit Values: WhatAmbTemp , temporarily return both thingys
*_________________________________________________________________________*/

int GetAmbTemp()

{
    int AmbTempTens;
    int AmbTempOnes;
    int result;

    /* read Temperature info from status string */
    AmbTempTens = ( ups->FStrBuff[64] );
    AmbTempOnes = ( ups->FStrBuff[65] );
    result = ( (AmbTempTens-48)*10 + (AmbTempOnes-48) );
    return result;
    //return (AmbTempOnes);

}
```

(iii) System Code Call: Mail Loop

```
/* Check the temperature */
WhatAmbTemp = GetAmbTemp();
sprintf(errorbuf," Temp OK on UPS1 ");
sprintf(msgbuf,"%d is UPS1 temp.",WhatAmbTemp);
if ( WhatAmbTemp > 23 )
{
    sprintf(errorbuf,"  UPS1 TEMP ALARM !!!  ");
    TempStamp("/tmp/UPStemp",errorbuf,msgbuf);
    MailTimer--;
    if ( MailTimer == 0 )
    {
    system( "mailx -s UPS1_Alarm checkups < /tmp/UPStemp" );
    /*sprintf(errorbuf," UPS1 TEMP ALARMed and mailed !");
    *TempStamp("/tmp/UPStemp",errorbuf,msgbuf);
    */
    MailTimer = 720;
    }
}
else
{
    TempStamp("/tmp/UPStemp",errorbuf,msgbuf);
}
```

(iv) The TempStamp( ) Function

```
/*****************************************************************************
* Function Name: TempStamp()
*
* Author(s): Doug and Barb
*
* Description: Enters the temperature from UPS1 into a file
*
* Entry Values: logname = Path and filename of the logfile to be updated
*               logtxt 1&2  = Text string to be entred into the log
*
* Exit Values: NONE
*____________________________________________________________________________*/

#ifdef USE_PROTOTYPES
void TempStamp(char *logname, char *logtext1, char *logtext2)
#else
TempStamp(logname, logtext1, logtext2)
char *logname;
char *logtext1;
char *logtext2;
#endif
{
int j;
```

```
    FILE *fp; /* Pointer to the log file handle */
    char cmdbuf[256];

    sprintf(cmdbuf,"date > %s",logname);
    system(cmdbuf);
    if ( fp = fopen(logname,"a") )
    {
        fprintf(fp,"%s\n",logtext1);
        fprintf(fp,"%s\n",logtext2);
        fputs("--------------------------------------------------------------------------------\n",fp);
        fclose(fp);
    }
    else
    {
        printf("CheckUPS Error: Could not open log file %s.\n The log entry has been skipped.\n",logname);
    }
}
```

Appendix D:

Output Code for the sprintf command

type	Output	Example
c	Character	**a**
d or **i**	Signed decimal integer	**392**
e	Scientific notation (mantise/exponent) using **e** character	**3.9265e2**
E	Scientific notation (mantise/exponent) using **E** character	**3.9265E2**
f	Decimal floating point	**392.65**
g	Use shorter %e or %f	**392.65**
G	Use shorter %E or %f	**392.65**
o	Signed octal	**610**
s	String of characters	**sample**
u	Unsigned decimal integer	**7235**
x	Unsigned hexadecimal integer	**7fa**
X	Unsigned hexadecimal integer (capital letters)	**7FA**
p	Address pointed by the argument	**B800:0000**
n	Nothing printed. The argument must be a pointer to integer where the number of characters written so far will be stored.	

Appendix E:

MRTG & Python Code

(i) MRTG Code

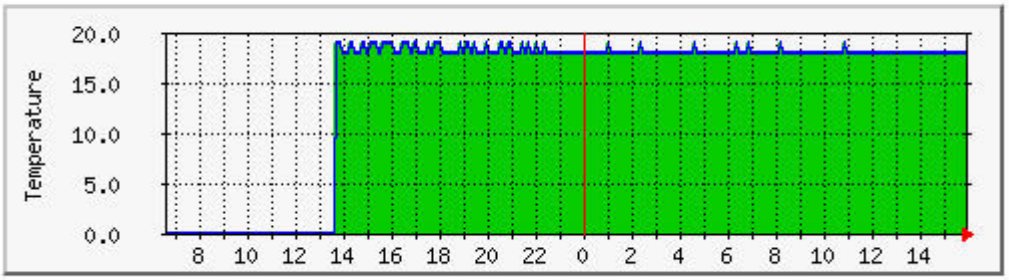
```
######################################################################
# System:
# Description: Ferrups 12.5KVA
# Contact:
# Location:
######################################################################

### UPS Temp >> Descr: 'UPS Temp '

Target[ups_temp]: `/usr/local/bin/gettemp.py`
Options[ups_temp]: gauge,growright
SetEnv[ups_temp]: MRTG_INT_IP="" MRTG_INT_DESCR="Passport"
MaxBytes[ups_temp]: 100
Title[ups_temp]: Analysis for UPS Temperature
PageTop[ups_temp]: <H1>Analysis for UPS Temperature
#Unscaled[ups_temp]: ymwd
ShortLegend[ups_temp]: deg
#XSize[ups_temp]: 380
#YSize[ups_temp]: 100
YLegend[ups_temp]: Temperature
Legend1[ups_temp]: Temperature
Legend2[ups_temp]: Temperature
Legend3[ups_temp]:
Legend4[ups_temp]:
LegendI[ups_temp]:
LegendO[ups_temp]:  Celsius; 
```

(ii) MRTG Daily graph

`Daily' Graph (5 Minute Average)



Max Celsius; 19.0 deg Average Celsius; 18.0 deg Current Celsius; 18.0 deg

(iii) Python Code

```
#! /usr/bin/env python

f=open('/var/log/local1.log', 'r')
f.seek(-17,2)
c = f.read(2)
f.close()
print c
print c
print "a while ;) "
print "UPS1"
```