Running a TAPR TICC with a Raspberry Pi

W.J. Riley
Hamilton Technical Services
Beaufort, SC 29907 USA
bill@wriley.com

• Introduction

This paper describes how the TAPR TICC time interval counter can be run with a Raspberry Pi computer and some custom software.

The TAPR TICC is a small, high-resolution time interval counter suitable for measuring the relative phase between two precision frequency sources based on a pair of the TAPR TADD-2 modules and a TICC module [1, 2]. The two TADD-2s are small 10 MHz to 1 pps dividers while the TICC is a daughter board mounted on an Arduino microcontroller that makes precise time interval measurements. The device can compare the relative phase between two 10 MHz sinewave inputs with a resolution below 100 ps.

The Raspberry Pi 3 Model B (RPi) [3] is a small and low cost Linux computer that can be run "headless" without a monitor, keyboard or mouse via its built-in Wi-Fi interface and an `SSH` connection. As such, it is an ideal way to capture data from the TICC without tying up a more expensive "real" computer.

Operation of the TICC/RPi clock measurement system can be supported by a custom Linux interface program that connects to the TICC through a virtual COM port, receives the TICC data stream, and captures it in the form of MJD timetagged phase data to a file and an optional PostgreSQL database. The latter is a particularly convenient way to archive, monitor and retrieve the clock data at LAN-connected workstations.

• User Interface Software

The TICC phase data can be captured to a file using the Raspberry Pi with an ordinary terminal program such as `GtkTerm`. However it is more convenient to use a custom user interface that can not only set the COM port and data filename, but also supports downsampling the phase data to a longer averaging time, monitoring the data stream, and storing it to a database for archiving and retrieval. A native Linux program called `TestDB` was developed for that purpose whose C source code is shown in Appendix I. It can be edited on the Raspberry Pi using the `Geany` IDE and built with the `gcc` makefile shown in Appendix II. The program uses the configuration file shown in Appendix III. This source code is made freely available under the license terms of Appendix IV, and it can be adapted to your particular requirements. In particular, the PostgreSQL database code, which uses the `libpq` C library, can be removed or revised for your particular database schema. Information about the PostgreSQL database used herein can be found in Reference [4]. Note that the `TestDB` program can be run using the database without writing data to disk (SD card) by entering the Linux null device `(/dev/null)` as the data file name.

The `TestDB` files are available at www.github.com/k2hrt/TestDB.

- **Running TestDB**

The `TestDB` program can be run remotely in the usual way through an `SSH` connection using `Putty` between a workstation and the RPi. An important difference, however, is that we want the TICC data capture to continue indefinitely after the SSH connection is closed, and be able to reestablish the connection to end the measurement run at some later time. Fortunately, there is a mechanism called `tmux` for doing that.

One can install `tmux` on the RPi with the command:

```
sudo apt-get install tmux
```

Then, after establishing an `SSH` connection to the RPi, and moving to the `TestDB` executable folder, one starts a `tmux` session with the command:

```
tmux
```

and begins the TICC measurements. The `SSH` connection can then be closed and the measurements will continue. Later, when it is time to end the measurement run, one reconnects via `SSH` to the `tmux` session with the command:

```
tmux attach –t #
```

where # is your session number (which starts at 0). Multiple sessions are supported, and can be listed with the command:

```
tmux list-sessions
```

So, to summarize, using `TestDB` along with `SSH` and `tmux`, one can perform a "headless" clock measurement with a TICC and Raspberry Pi that can be controlled from a remote workstation and optionally monitored via a database server to provide excellent resolution, flexibility and convenience at exceptionally low cost.

- **UPS**

The Raspberry Pi (and the rest of the clock measuring system) should be run from an uninterruptable power supply not only for measurement continuity but also to avoid non-graceful RPi shutdown which can cause SD card corruption.

- **Endurance and Life**

An SD card flash memory has limited write cycle endurance. A typically-cited value is 100,000 cycles without "wear leveling" [5], which extends it by spreading out the write cycles over the storage media. At a 100 second measurement tau, this non-leveled endurance limit could be exceeded in only about 100 days (most memory cells don't change during each measurement). High-endurance cards (e.g., 2 million cycle) are available and recommended, as is using a larger-capacity card. That is a strong incentive to (a) use an external database to store the measurement data, (b) don't store data on the SD card, especially at a short measurement tau, (c) attach an external USB hard drive for data storage, and

(d) have a spare SD card with the operating system and critical files (standard software can be re-installed). See Reference [6] & [7] for more suggestions.

An SD card also has limited data storage life, often stated as 5 years (high temperatures should be avoided). Presumably any important data will be downloaded from the RPi before that becomes an issue. Copies of critical files should be kept on another type of storage media (e.g., magnetic or optical disk).

- **References**

1. TADD-2 Assembly and Operation Manual, Tucson Amateur Packet Radio Corporation (TAPR), July 2009.
2. TAPR TICC Timestamping Counter Operation Manual, Tucson Amateur Packet Radio Corporation (TAPR), March 2017.
3. The Raspberry Pi Foundation.
4. W.J. Riley, "A PostgreSQL Database for the PicoPak Clock Measurement Module".
5. See: https://en.wikipedia.org/wiki/Wear_leveling.
6. See: https://raspberrypi.stackexchange.com/questions/169/how-can-i-extend-the-life-of-my-sd-card.
7. See: http://www.makeuseof.com/tag/extend-life-raspberry-pis-sd-card/.

File: Running the TAPR TICC with a Raspberry Pi.doc

W.J. Riley
Hamilton Technical Services

March 25, 2017
Rev A May 5, 2017
Rev B May 6, 2017
Rev C July 15, 2017

Appendix I – TestDB C Source Code

```
// TestDB
// W.J. Riley, K2HRT
// 03/24/17
// Last Revised 07/14/17
// Program to test a simple interface between
// the TAPR TICC and a PostgreSQL database
// to store 1 pps clock measurements
// It uses the PicoPak database at 192.168.2.40
// on the local LAN, clock and reference ids=7,
// a module S/N=1001, and a measurement description
// of TAPR TICC Clock Data

// This test program uses default database parameters
// except that it uses the next meas_id

// The TICC should be outputing its 1s data stream
// before launching the program.  The 1st datum
// received may be incomplete and is ignored

// The TICC normally resets when a new connection is made
// and displays a startup message with a delay
// for the operator to enter a setup menu
// That behavior must be disabled by installing JP1

// The program makes a connection to the database,
// gets the next meas_id, enters the measurement run
// into the database, opens the TICC COM port,
// swallows the first reading, and enters an endless loop
// to the get the TICC data, get the MJD timetag
// and store them in the database

// The TICC.ini file is read if READINI macro=1
// That file can be edited to change measurement settings

// The TICC module must be manually entered into
// the measurement_modules database table

// Postgresql header files
#include </usr/include/postgresql/libpq-fe.h>
#include </usr/include/postgresql/postgres_ext.h>

// Standard header files
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

// File and port header files
#include <sys/fcntl.h> // For open(), etc.
#include <unistd.h>  // For UNIX definitions
#include <termios.h>  // For port settings

// UNIX time header
#include <sys/timeb.h>

// Terminal headers
#include <sys/select.h>
#include <termios.h>
#include <stropts.h>
#include <sys/ioctl.h>

// Size macros
```

```
#define BUFFER_SIZE 128
#define QUERY_SIZE 256

// Parameter default macros
#define HOST "192.168.2.40"
#define DATABASE "ppd"
#define USER "postgres"
#define PASSWORD "root"
#define FILENAME "/home/bill/Data/TICC.dat"
#define INIFILE "TICC.ini"
#define SIG_ID 7
#define REF_ID 7
#define DESCRIP "TAPR TICC Clock Data"
#define SN 1003
#define FREQ 1e7
#define TAU 1
#define AF 1

// Control macros
#define VERBOSE 1
#define READINI 1
#define USEDB 1

// Function prototypes
PGconn *ConnectToDB(char *database, char *user, char *host, char *password);
void CloseDB(PGconn *conn, int sn, int nmeas);
int OpenPort(void);
double ReadCOM(int fd);
double GetMJD(void);
int StoreMeasurement(PGconn *conn, int sn, double timetag, double meas);
PGresult *DoSQL(PGconn *conn, char *cmd);
PGresult *DoSQL(PGconn *conn, char *cmd);
int DoQuotes(char *s);
int GetNextMeasID(PGconn *conn);
int EnterMeas(PGconn *conn, int meas_id, int sn, int sig_id, int ref_id,
      double frequency, char *description, double begin_mjd, double tau);
int _kbhit(void);
int OpenFile(char *filename);
void EnterParams(char *description, char *filename, int *sig_id, int *ref_id, int *sn,
      double *tau, int *af, double *frequency, char *host, char *database, char *user, char
*password);
int ReadIniValue(char *key, char *value);
int GetParams(char *description, char *filename, int *sig_id, int *ref_id, int *sn,
      double *tau, int *af, double *frequency, char *host, char *database, char *user, char
*password);

// Main function to test database access
int main(void)
{
      // Local variables
      PGconn *conn=0;
      char c; // Keyboard entry
      char description[BUFFER_SIZE+1]; // Measurement decription
      char host[BUFFER_SIZE+1]; // Host IP address
      char database[BUFFER_SIZE+1]; // Database name
      char user[BUFFER_SIZE+1]; // User name
      char password[BUFFER_SIZE+1]; // User password
      char filename[BUFFER_SIZE+1]; // Data filename
      int fd=0; // File descriptor
      int sig_id=SIG_ID; // Signal source ID #
      int ref_id=REF_ID; // Refeence source ID #
      int meas_id=0; // Measurement run #
      int sn=SN; // Module S/N
      int af=1; // Averaging factor
      int db_connected=0; // Database connection flag
      int file_open=0; // Data file open flag
```

```
int nmeas=0; //  # measurenments
double phase=0; // Phase value
double mjd=0; // Current MJD
double frequency=FREQ; // Nominal frequency
double tau=TAU; // Measurement tau
FILE *fptr; // File pointer

// Set default description
strcpy(description, DESCRIP);

// Set default host
strcpy(host, HOST);

// Set default database
strcpy(database, DATABASE);

// Set default username
strcpy(user, USER);

// Set default password
strcpy(password, PASSWORD);

// Set default filename
strcpy(filename, FILENAME);

// Show introduction\n"
printf("TAPR TICC Data Capture Program\n");
printf("TestDB Version 0.01 of 03/22/17\n");
printf("W.J. Riley, K2HRT, bill@wriley.com\n");
printf("Respond with Enter to accept parameter values(s)\n");
printf("or initial letter to change a parameter\n");
printf("x will end all parameter entries\n");
printf("q will end measurements and close program\n\n");

if(READINI)
{
       // Get measurement parameters from ini file
       GetParams(description, filename, &sig_id, &ref_id, &sn,
              &tau, &af, &frequency, host, database, user, password);
}

// Enter measurement parameters
EnterParams(description, filename, &sig_id, &ref_id, &sn,
       &tau, &af, &frequency, host, database, user, password);

// Open data file
// We support only write mode, not append
fptr=fopen(filename, "w");
if((fptr==NULL))
{
       // Error opening file
       printf("Unable to open file %s\n", filename);
}
else
{
       // File opened OK
       printf("File %s opened\n", filename);
       file_open=1;
}

if(USEDB)
{
       // Connect to database
       conn=ConnectToDB(DATABASE, USER, HOST, PASSWORD);

       // Was a database connection made?
```

```
		if(conn)
		{
			// Set database connected flag
			db_connected=1;
		}

		// Get next meas_id
		meas_id=GetNextMeasID(conn);

		#if(1) // For testing
		printf("meas_id=%d\n", meas_id);
		#endif

		// Enter measurement run into database
		EnterMeas(conn, meas_id, sn, sig_id, ref_id, frequency, description, mjd,
tau*af);
	}

	// Open COMport
	fd=OpenPort();

	#if(0) // For testing
	// This message could be retained
	printf("File descriptor=%d\n", fd);
	#endif

	// Initialize nmeas to skip first few points
	// that may be bad or duplicates
	nmeas=-9;

	#if(1) // Main loop
	// Read and store timetagged TICC phase measurements
	// Loop forever until q command received
	while(1)
	{
		// Get phase reading from TICC
		phase=ReadCOM(fd);

		// Check for bad reading
		// Value must be between +/- 1.0 and not zero
		if((phase!=0.0)&&(phase>=-1.0)&&(phase<=1.0))
		{
			// Get current MJD from system
			mjd=GetMJD();

			#if(1) // For testing
			// This message can be retained
			// Meas count is <1 while points being skipped
			// MJD is displayed as 12345.123456
			// Phase data is of form -0.123456789012
			printf("#=%d, MJD=%12.6f, Phase=%15.12f\n", nmeas, mjd, phase);
			#endif

			if(db_connected)
			{
				// Skip 1st few points
				// They may be bad or duplicates
				if((nmeas>0))
				{
					if(((nmeas%af)==0))
					{
						// Store measurement in database
						StoreMeasurement(conn, sn, mjd, phase);

						#if(1) // For testing
						// This message should be retained
```

```
                                        printf("Meas stored in database: Pt #=%d\n",
nmeas/af);
                                        #endif
                                }
                        }
                }

                if((nmeas>0))
                {
                        if(((nmeas%af)==0))
                        {
                                // Store measurement in file
                                // They may be bad or duplicates
                                fprintf(fptr, "%12.6f %15.12f\n", mjd, phase);

                                #if(1) // For testing
                                // This message should be retained
                                printf("Meas stored in file: Pt #=%d\n", nmeas/af);
                                #endif
                        }
                }

                // Increment mesurement count
                nmeas++;
        }

        // Check for quit command
        if(_kbhit())
        {
                if(((c=getchar())=='q'))
                {
                        break;
                }
        }
    }
    #endif

    // Close COM port
    close(fd);

    // Close result struct
    // PQclear(res);

    // Put end MJD into database
    // Set TICC module inactive in database
    // Close database connection
    if(db_connected)
    {
            CloseDB(conn, sn, nmeas);
    }

    // Close data file
    if(file_open)
    {
            fclose(fptr);
    }

    return 0;
}

#if(1)
// Function to connect to PostgreSQL database
// Pass connection parameters
// Returns pointer to PostgreSQL connection struct
// and sets db_connected flag if connection made
// or returns 0 if not connected
```

```
PGconn *ConnectToDB(char *database, char *user, char *host,
	char *password)
{
	// Local variables
	// Must insure no buffer overflow, no truncation and proper EOS
	// termination with database, user, host and password each having
	// maximum lengths of BUFFER_SIZE (not counting their EOS at end)
	// So conninfo has a maximum length of 4*BUFFER_SIZE
	// plus the dbname= user= hostaddr= password= characters and EOS
	// Buffer overflow can be avoided by making conninfo 5*BUFFER_SIZE
	char conninfo[5*BUFFER_SIZE]; // Connection parameter text buffer
	PGconn *conn; // Database connection structure pointer

	// Make connection to database
	// Copy connection parameters into conninfo
	// A blank host IP address will connect to localhost
	// The localhost IP adr 127.0.0.1 may also be used

	if(strlen(host)==0)
	{
		// No host address - leave it out - connect to localhost
		sprintf(conninfo, "dbname=%s user=%s password=%s",
			database, user, password);
	}
	else
	{
		// Include host address

		sprintf(conninfo, "dbname=%s user=%s hostaddr=%s password=%s",
			database, user, host, password);
	}

	// Connect to PostgreSQL server
	conn = PQconnectdb(conninfo);

	// Check that the connection was successfully made
	if (PQstatus(conn) != CONNECTION_OK)
	{
		#if(1) // For testing
		printf("Unable to connect to database\n");
		#endif

		// Return error code
		return 0;
	}

	#if(1) // For testing
	// This message should be retained
	printf("Connected to database\n");
	#endif

	// Return pointer to PostgreSQL connection structure
	return conn;
}
#endif

#if(1)
// Function to close connection to database
// Enter end MJD into measirement_list table
// Set the TICC module inactive
// Close database connection
void CloseDB(PGconn *conn, int sn, int nmeas)
{
	// Local variables
	char query[QUERY_SIZE+1]; // PostgreSQL query text buffer
	PGresult *res; // Query result pointer
```

```
	double mjd=0; // End MJD

	// Get end MJD
	mjd = GetMJD();

	// Compose PostgreSQL command to put end MJD into measurement list
	sprintf(query, "UPDATE measurement_list SET end_mjd=%12.6f WHERE meas_id=%d",
		mjd, nmeas);

	// For testing
	// printf("%s\n", query);

	// Execute command
	if ((res = PQexec(conn, query)) == 0)
	{
		// PQexec() only rarely returns a NULL pointer
		// If so, display error message
		printf("Error3 entering measurement into database");
	}
	else // Valid res pointer
	{
		// Check status of database INSERT command
		if (PQresultStatus(res) != PGRES_COMMAND_OK)
		{
			// Status should be PGRES_COMMAND_OK
			// If not, display error message
			printf("Error4 entering measurement into database");
		}
	}

	// We also want to set the module inactive in its measurement_modules table
	// using UPDATE measurement_modules SET active=FALSE WHERE sn=nSN

	// Compose PostgreSQL command
	sprintf(query, "UPDATE measurement_modules SET active=FALSE WHERE sn=%d", sn);

	// For testing
	// printf("%s\n", query);

	// Execute command
	if ((res = PQexec(conn, query)) == 0)
	{
		// PQexec() only rarely returns a NULL pointer
		// If so, display error message
		printf("Error1 setting module inactive in database");
	}
	else // Valid res pointer
	{
		// Check status of database INSERT command
		if (PQresultStatus(res) != PGRES_COMMAND_OK)
		{
			// Status should be PGRES_COMMAND_OK
			// If not, display error message
			printf("Error2 setting module inactive in database");
		}
	}

	// Close database connection
	PQfinish(conn);

	#if(1) // For testing
	{
		printf("Database connection closed\n");
	}
	#endif
}
```

```
#endif

#if(1)
// Function to open virtual serial COM port
// The TICC COM port is /dev/ttyACM0
// with port settings 115,200 Baud 8N1
// Return file descriptor
int OpenPort(void)
{
	// Local variables
	struct termios tc; // Port settings struct
	int fd=0; // File descriptor

	// Open port
	fd=open("/dev/ttyACM0", O_RDONLY | O_NOCTTY);

	// Get current port options
	tcgetattr(fd, &tc);

	// Set Baud rates to 115,200
	cfsetispeed(&tc, B115200);
	cfsetospeed(&tc, B115200);


	// Set local mode & enable receiver
	tc.c_cflag |= (CLOCAL | CREAD);

	// Set the new port options now
	tcsetattr(fd, TCSANOW, &tc);

	// Set # data bits
	tc.c_cflag &= ~CSIZE; // Mask character size bits
	tc.c_cflag |= CS8;    // Select 8 data bits

	// Set 8 data bits, no parity, 1 stop bit (8N1)
	tc.c_cflag &= ~PARENB;
	tc.c_cflag &= ~CSTOPB;
	tc.c_cflag &= ~CSIZE;
	tc.c_cflag |= CS8;

	// Use raw output
	tc.c_cflag &= ~OPOST;

	// Set no hardware flow cpntrol
	tc.c_cflag &= ~CRTSCTS;

	return fd;
}
#endif

#if(1) // New
// Read COM Port
// Get one TICC phase measurement
// Revised 07/13/17 to handle negative phase values
double ReadCOM(int fd)
{
	// Local variables
	char input[BUFFER_SIZE]; // Read buffer
	char value[BUFFER_SIZE+1]; // Value buffer
	int n=0; // # chars read - used only for testing
	double phase=0.0; // Phase value

	// Initialize the input buffer
	strcpy(input, "");

	// Initialize the value buffer
```

```
	strcpy(value, "");

	// Read TICC data stream 1 character at a time
	// into input buffer buffer
	// and concatenate those characters into value buffer

	// Read 1 line of input from TICC
	while(input[0]!='\n')
	{
		// Read incoming character
		read(fd, input, 1);

		// Truncate the input buffer
		input[1]='\0';

		// Is character numeric?
		if((isdigit(input[0])) || (input[0]=='-') || (input[0]=='.'))
		{
			// Add character to value buffer
			strcat(value, input);

			// Increment the character count
			n++;

			#if(0) // For testing
			// Display the character and value string
			// The value string grows to something like:
			// -0.123456789012-
			printf("%i, %c, %s\n", n, input[0], value);
			#endif
		}
	}

	// Add EOS after phase value
	// Discard the trailing - at the end from the TI A->B
	value[15]='\0';

	// Get phase value
	phase=atof(value);

	#if(0) // For testing
	// Show the phase string and value
	printf("Value=%s, ", value);
	printf("Phase=%15.12f\n", phase);
	#endif

	return phase;
}
#endif

#if(1)
// Function to get current MJD
// System should use NTP to set its clock
// Use this version which has high resolution (1ms)
double GetMJD(void)
{
	// Local variables
	double mjd; // MJD
	double sec; // UNIX seconds
	double millisec; // UNIX milliseconds
	struct timeb tb; // UNIX time strut

	// Get MJD from system clock
	ftime(&tb);
	sec=(double)(tb.time);
	millisec=(double)(tb.millitm);
```

```
	mjd=40587.0+(sec/86400.0)+(millisec/(86400.0*1000.0));

	#if(0) // for testing
	printf("sec=%f\n", sec);
	printf("millisec=%f\n", millisec);
	printf("mjd=%f\n", mjd);
	#endif

	return mjd;
}
#endif

#if(0)
// Function to get current MJD
// System should use NTP to set its clock
// This version has low resolution (1s)
// Use version above with higher resolution
double GetMJD(void)
{
	// Local variables
	double mjd; // MJD
	double sec; // UNIX seconds

	// Get MJD from system clock
	sec=(double)(time(NULL));
	mjd=40587.0+(sec/86400.0);

	#if(1) // For testing
	printf("time=%lu\n", time(NULL));
	printf("sec=%f\n", sec);
	printf("days=%f\n", sec/86400.0);
	printf("mjd=%f\n", mjd);
	#endif

	return mjd;
}
#endif

#if(1)
// Function to store timetagged measurement in PostgreSQL database
// Pass database connection strut, MJD timetag and phase measurement
// Return success or error codeint GetNextMeasID(void)
int StoreMeasurement(PGconn *conn, int sn, double timetag, double meas)
{
	// Local variables
	// PG_QUERY_SIZE is 256
	// sn has maximum size of (say) 6 digits (characters)
	// mjd timetag XXXXX.XXXXXX has 12 characters
	// meas has 16.12e format X.XXXXXXXXXXXXe-XX 18 characters
	// rest of command has about 50 characters
	// so szQuery[] has plenty of room to avoid buffer overflow
	char query[QUERY_SIZE+1];  // PostgreSQL query text buffer
	PGresult *res=NULL; // Query result pointter

	// Compose PostgreSQL command
	sprintf(query, "INSERT INTO measurements(sn, mjd, meas)"
		" VALUES(%d, %5.6f, %20.15e)", sn, timetag, meas);

	// For testing
	// printf(query);

	// Execute command
	if ((res = PQexec(conn, query)) == 0)
	{
		// PQexec() only rarely returns a NULL pointer
		// If so, display error message
```

```
			// Error should be handled by calling function

			#if(0) // For testing
			printf("Query failed");
			#endif

			// Return error code
			return 0;
		}
		else // Valid res pointer
		{
			// Check status of database INSERT command
			if (PQresultStatus(res) != PGRES_COMMAND_OK)
			{
				// Status should be PGRES_COMMAND_OK
				// If not, display error message
				// Error should be handled by calling function

				#if(0) // For testing
				printf("Query failed");
				#endif

				// Return error code
				return 0;
			}
		}

		// Return success code
		return 1;
}
#endif

#if(1)
// Function to execute PostgreSQL query
// Parameters are database connection struct and query
// Returns query result
PGresult *DoSQL(PGconn *conn, char *cmd)
{
		// Local variables
		// Error must hold the query plus a bit more
		char error[QUERY_SIZE+1]; // Error message
		PGresult *res; // Query result pointer

		// Execute query
		res = PQexec(conn, cmd);

		// Check status - trap fatal errors and bad responses
		if ((PQresultStatus(res) == PGRES_FATAL_ERROR) ||
			(PQresultStatus(res) == PGRES_BAD_RESPONSE))
		{
			// Check connection
			if ((PQstatus(conn) == CONNECTION_OK))
			{
				// Display query error message
				sprintf(error, "Database query \"%s\" failed.\n"
					"Program will close.", cmd);
				printf("%s/n", error);

				// Close down PostgreSQL
				PQclear(res);
				PQfinish(conn);

				// Return error code
				return 0;
			}
			else // Bad connection
```

```
		{
			// Try to reset connection?
			// Display bad connection error message
			sprintf(error, "Bad database connection.\n"
				"Program will close.");
			printf("%s", error);

			// Close down PostgreSQL
			PQclear(res);
			PQfinish(conn);

			// Return error code
			return 0;
		}
	}

	// Query successful
	return res;
}
#endif

#if(1)
// Function to add single quotes around a string
int DoQuotes(char *s)
{
	// Local variables
	char b1[QUERY_SIZE+1]; // String buffer 1
	char b2[QUERY_SIZE+1]; // String buffer 2
	char *p; // String pointer

	// Copy s to b1
	strcpy(b1, s);

	// Initialize pointer to buffer 1
	p = b1;

	// Change all single quotes to excaped ones
	while((p=(strstr(p, "'")))	// Keep looking until no more ' characters
	{
		*p = '\0';		// Replace ' with EOS
		strcpy(b2, b1);		// Copy first part of b1 to b2
		strcat(b2, "''");   // Concatenate '' to b2 // Was "\\'"
		p++;			// Advance pointer to next character of b1
		strcat(b2, p);		// Concatenate last part of b1 to b2
		strcpy(b1, b2);		// Copy b2 back to b1
		p++;			// Advance pointer to next character of b1
	}

	// Copy single quote to s
	strcpy(s, "'");

	// Concatenate b1 to s
	strcat(s, b1);

	// Concatenate single quote to s
	strcat(s, "'");

	return strlen(s);
}
#endif

#if(1)
// Function to get next meas_id
int GetNextMeasID(PGconn *conn)
{
	// Local variables
```

```
	int meas_id; // Measurement ID
	char query[QUERY_SIZE]; // Query text
	PGresult *res=NULL; // Query result pointer

	// Get last meas_id by querying measurement_list table with:
	// SELECT meas_id FROM measurement_list ORDER BY meas_id DESC LIMIT 1
	// Compose query
	sprintf(query, "SELECT meas_id FROM measurement_list ORDER BY meas_id DESC LIMIT 1");

	// Perform query
	if ((res = DoSQL(conn, query)) == 0)
	{
		#if(1) // For testing
		// Display query error message
		printf("Error #1 reading meas_id from database\n");
		#endif

		// Return error code
		return 0;
	}

	// Make sure we had at least one clock_id
	if (PQntuples(res))
	{
		// Get and increment # measurements
		meas_id = atoi(PQgetvalue(res, 0, 0));
		meas_id++;

		#if(1) // For testing
		printf("Next meas_id=%d\n", meas_id);
		#endif
	}
	else // Error
	{
		#if(1) // For testing
		// Display query error message
		printf("Error #2 reading meas_id from database\n");
		#endif

		// Return error code
		meas_id=0;
	}

	// Return next meas_id, or 0 if error
	return meas_id;
}
#endif

#if(1)
// Function to enter measurement run into database
// Parameters to be entered are:
// meas_id, sn, sig_id, ref_id, frequency, description, begin_mjd, tau
// The 1st 4 are ints, description is a string, and the other 3 are doubles
// The function returns a success (1) or error (0) code
int EnterMeas(PGconn *conn, int meas_id, int sn, int sig_id, int ref_id,
	double frequency, char *description, double begin_mjd, double tau)
{
	// Local variables
	char query[QUERY_SIZE]; // Query text
	char error[QUERY_SIZE+1]; // Error message
	PGresult *res=NULL; // Query result pointer

	// Put info about run into the measurement_list database table

	// Format description with single quotes
	// Note: DoQuotes() previously excaped embedded single quotes (') with (\')
```

```
	// That doesn't work right anymore in PostgreSQL 9.1: get Error2 below
	// New DoQuotes() excapes embedded single quotes (') with two single quotes ('')
	// That apparently does work
	DoQuotes(description);

	#if(1) // Already have current MJD
	// Get current mjd by calling GetMJD();
	// The MJDs here and after run is done will be slightly earlier and later
	// than the measurement timetags - should be OK for selecting data from database
	begin_mjd = GetMJD();
	#endif

	// PostgreSQL command to put everything except end_mjd into measurement_list table
	// INTERT INTO measurement_list(meas_id, sn, sig_id, ref_id, frequency, description,
begin_mjd, tau)
	// VALUES(1, nSN, nSig, nRef, fNom, szDescription, fMJD, fTau)
	// meas_id, sn, sig_id, & ref_id are integers
	// frequency and tau are reals (doubles)
	// begin_mjd and end_mjd are numeric (doubles)
	// description is a character string (with single quotes)
	// tau is always 1 second

	// Compose PostgreSQL command
	sprintf(query, "INSERT INTO measurement_list(meas_id, sn, sig_id, ref_id, frequency,
description, begin_mjd, tau)"
		" VALUES(%d, %d, %d, %d, %e, %s, %12.6f, %e)", meas_id, sn, sig_id, ref_id,
frequency, description, begin_mjd, tau);

	// Execute command
	if ((res = PQexec(conn, query)) == 0)
	{
		// PQexec() only rarely returns a NULL pointer
		// If so, display error message
		printf(error, "Error1 entering measurement into database");

		// Return error code
		return 0;
	}
	else // Valid res pointer
	{
		// Check status of database INSERT command
		if (PQresultStatus(res) != PGRES_COMMAND_OK)
		{
			// Status should be PGRES_COMMAND_OK
			// If not, display error message
			printf(error, "Error2 entering measurement into database");

			// Return error code
			return 0;
		}
	}

	// We also want to set the module active in its measurement_modules table
	// using UPDATE measurement_modules SET active=TRUE WHERE sn=nSN

	// Compose PostgreSQL command
	sprintf(query, "UPDATE measurement_modules SET active=TRUE WHERE sn=%d", sn);

	// Execute command
	if ((res = PQexec(conn, query)) == 0)
	{
		// PQexec() only rarely returns a NULL pointer
		// If so, display error message
		printf(error, "Error1 setting module active in database");

		// Return error code
```

```
		return 0;
	}
	else // Valid res pointer
	{
		// Check status of database INSERT command
		if (PQresultStatus(res) != PGRES_COMMAND_OK)
		{
			// Status should be PGRES_COMMAND_OK
			// If not, display error message
			printf(error, "Error2 setting module active in database");

			// Return error code
			return 0;
		}
	}

	// Return success code
	return 1;
}
#endif

#if(1)
// Linux (POSIX) implementation of _kbhit().
// Morgan McGuire, morgan@cs.brown.edu
int _kbhit(void)
{
    static const int STDIN = 0;
    static int initialized = 0;
    static struct termios term;
    int bytesWaiting;


    if(!initialized)
    {
        // Use termios to turn off line buffering
        tcgetattr(STDIN, &term);
        term.c_lflag &= ~ICANON;
        tcsetattr(STDIN, TCSANOW, &term);
        setbuf(stdin, NULL);
        initialized = 1;
    }

    // Get # chars in keyboard buffer
    ioctl(STDIN, FIONREAD, &bytesWaiting);

    return bytesWaiting;
}
#endif

#if(1)
int OpenFile(char *filename)
{
	// Local variables
	int open=0; // File opened flag

	return open;
}
#endif

#if(1)
// Function to enter TICC measurement parameters
// Note that all function parameters are pointers
void EnterParams(char *description, char *filename, int *sig_id, int *ref_id, int *sn,
	double *tau, int *af, double *frequency, char *host, char *database, char *user, char
*password)
{
```

```
// Local variables
char c; // Keyboard entry
int done=0; // Entry flag
int alldone=0;      // All entries done flag

// Parameter entries are in order of most likely to change
// Can end parameter entry with 'x' after any one done

// Begin measurement description entry
// Show/Enter measurement description
printf("Measurement run description:\n");
printf("Description=%s\n", description);

// Wait for user entry
while(!_kbhit());

#if(0) // For testing
printf("  Key=%d\n", getchar());
#endif

// Are entries continuing?
if(!alldone)
{
       // Wait for user entry
       while(!_kbhit());

       #if(0) // For testing
       printf("  Key=%d\n", getchar());
       #endif

       // Perform description entry
       while((c=getchar()))
       {
              // Parse description entry
              switch(c)
              {
                     // Description
                     case 'd':
                     case 'D':
                     {
                            printf("=Enter new description:\n");
                            scanf("%s", description);
                            getchar(); // Swallow Enter
                            printf("Description=%s", description);
                     }
                     break;

                     // Enter
                     case '\n':
                     {
                            #if(0) // For testing
                            printf("Enter\n\n");
                            #endif

                            // Set done flag
                            done=1;
                     }
                     break;

                     // All done
                     case 'x':
                     {
                            #if(1) // For testing
                            printf("=All done\n");
                            #endif
```

```
                        // Set done flags
                        done=1;
                        alldone=1;
                }
        }

        // Value accepted
        if(done)
        {
                // Reset done flag
                done=0;

                // Done with this parameter
                break;
        }
    }
}

// Are entries continuing?
if(!alldone)
{
    // Begin filename entry
    // Show/Enter filename
    printf("Data filename:\n");
    printf("Filename=%s\n", filename);

    while((c=getchar()))
    {
        // Parse filename entry
        switch(c)
        {
                // Description
                case 'f':
                case 'F':
                {
                        printf("=Enter new file name:\n");
                        scanf("%s", filename);
                        getchar(); // Swallow Enter
                        printf("File name=%s", filename);
                }
                break;

                // Enter
                case '\n':
                {
                        #if(0) // For testing
                        printf("Enter\n\n");
                        #endif

                        // Set done flag
                        done=1;
                }
                break;

                // All done
                case 'x':
                {
                        #if(1) // For testing
                        printf("=All done\n");
                        #endif

                        // Set done flags
                        done=1;
                        alldone=1;
                }
        }
```

```
		// Value accepted
		if(done)
		{
			// Reset done flag
			done=0;

			// Done with this parameter
			break;
		}
	}
}

// Are entries continuing?
if(!alldone)
{
	// Begin clock entries
	// Show/Enter clocks
	printf("Signal and reference clocks:\n");
	printf("Signal ID=%d, Reference ID=%d\n", *sig_id, *ref_id);

	// Wait for user entry
	while(!_kbhit());

	#if(0) // For testing
	printf("  Key=%d\n", getchar());
	#endif

	// Perform clock entry
	while((c=getchar()))
	{
		// Parse description entry
		switch(c)
		{
			// Signal clock
			case 's':
			case 'S':
			{
				printf("=Enter new signal clock ID #:\n");
				scanf("%d", sig_id);
				getchar(); // Swallow Enter
				printf("Signal ID=%d, Reference ID=%d\n", *sig_id, *ref_id);
			}
			break;

			// Reference clock
			case 'r':
			case 'R':
			{
				printf("=Enter new reference clock ID #:\n");
				scanf("%d", ref_id);
				getchar(); // Swallow Enter
				printf("Signal ID=%d, Reference ID=%d\n", *sig_id, *ref_id);
			}
			break;

			// Enter
			case '\n':
			{
				#if(0) // For testing
				printf("Enter\n\n");
				#endif

				// Set done flag
				done=1;
			}
```

```
                        break;

                        // All done
                        case 'x':
                        {
                                #if(1) // For testing
                                printf("=All done\n");
                                #endif

                                // Set done flags
                                done=1;
                                alldone=1;
                        }
                }

                // Value accepted
                if(done)
                {
                        // Reset done flag
                        done=0;

                        // Done with these parameters
                        break;
                }
        }
}

// Are entries continuing?
if(!alldone)
{
        // Begin module S/N entry
        // Show/Enter module
        printf("TICC Module S/N:\n");
        printf("Module S/N=%d\n", *sn);

        // Wait for user entry
        while(!_kbhit());

        #if(0) // For testing
        printf("  Key=%d\n", getchar());
        #endif

        // Perform module S/N entry
        while((c=getchar()))
        {
                // Parse module entry
                switch(c)
                {
                        // Description
                        case 'm':
                        case 'M':
                        {
                                printf("=Enter new module S/N:\n");
                                scanf("%s", description);
                                getchar(); // Swallow Enter
                                printf("sn=%s", description);
                        }
                        break;


                        // Enter
                        case '\n':
                        {
                                #if(0) // For testing
                                printf("Enter\n\n");
                                #endif
```

```
                        // Set done flag
                        done=1;
                }
                break;

                // All done
                case 'x':
                {
                        #if(1) // For testing
                        printf("=All done\n");
                        #endif

                        // Set done flags
                        done=1;
                        alldone=1;
                }
        }

        // Value accepted
        if(done)
        {
                // Reset done flag
                done=0;

                // Done with this parameter
                break;
        }
    }
}

// Are entries continuing?
if(!alldone)
{
    // Begin tau entries
    // Show/Enter tau and af
    printf("Measurement tau and averaging factor:\n");
    printf("Tau=%e, AF=%d\n", *tau, *af);

    // Wait for user entry
    while(!_kbhit());

    #if(0) // For testing
    printf("  Key=%d\n", getchar());
    #endif

    // Perform tau andb af entries
    while((c=getchar()))
    {
        // Parse description entry
        switch(c)
        {
                //
                case 't':
                case 'T':
                {
                        printf("=Enter new measurenment tau:\n");
                        scanf("%lf", tau);
                        getchar(); // Swallow Enter
                        printf("Tau=%e, AF=%d\n", *tau, *af);
                }
                break;

                // Averaging Factor
                case 'a':
                case 'A':
```

```
                        {
                                printf("=Enter new averaging factor:\n");
                                scanf("%d", af);
                                getchar(); // Swallow Enter
                                printf("Tau=%f, AF=%d\n", *tau, *af);
                        }
                        break;

                        // Enter
                        case '\n':
                        {
                                #if(0) // For testing
                                printf("Enter\n\n");
                                #endif

                                // Set done flag
                                done=1;
                        }
                        break;

                        // All done
                        case 'x':
                        {
                                #if(1) // For testing
                                printf("=All done\n");
                                #endif

                                // Set done flags
                                done=1;
                                alldone=1;
                        }
                }

                // Value accepted
                if(done)
                {
                        // Reset done flag
                        done=0;

                        // Done with these parameters
                        break;
                }
        }
}

// Are entries continuing?
if(!alldone)
{
        // Begin frequency entry
        // Show/Enter frequency
        printf("Nominal frequency:\n");
        printf("Frequency=%lf\n", *frequency);

        while((c=getchar()))
        {
                // Parse frequency entry
                switch(c)
                {
                        // Description
                        case 'f':
                        case 'F':
                        {
                                printf("=Enter new frequency:\n");
                                scanf("%lf", frequency);
                                getchar(); // Swallow Enter
                                printf("Frequency=%lf", *frequency);
```

```
                        }
                        break;

                        // Enter
                        case '\n':
                        {
                                #if(0) // For testing
                                printf("Enter\n\n");
                                #endif

                                // Set done flag
                                done=1;
                        }
                        break;

                        // All done
                        case 'x':
                        {
                                #if(1) // For testing
                                printf("=All done\n");
                                #endif

                                // Set done flags
                                done=1;
                                alldone=1;
                        }
                }

                // Value accepted
                if(done)
                {
                        // Reset done flag
                        done=0;

                        // Done with this parameter
                        break;
                }
        }
}

// Are entries continuing?
if(!alldone)
{
        // Begin database entries
        // Show database credentials
        printf("Database credentials:\n");
        printf("Host=%s, Database=%s, User=%s, Password=%s\n",
                host, database, user, password);

        // Wait for user entry

        while(!_kbhit());

        #if(0) // For testing
        printf("  Key=%d\n", getchar());
        #endif

        // Perform database credential entry
        while((c=getchar()))
        {
                // Parse database credential entry
                switch(c)

                {
                        // Host IP address
                        case 'h':
```

```
case 'H':
{
	printf("=Enter new host IP address:\n");
	scanf("%s", host);
	getchar(); // Swallow Enter
	printf("Host=%s, Database=%s, User=%s, Password=%s\n",
		host, database, user, password);
}
break;

// Database name
case 'd':
case 'D':
{
	printf("=Enter new database name:\n");
	scanf("%s", database);
	getchar(); // Swallow Enter
	printf("Host=%s, Database=%s, User=%s, Password=%s\n",
		host, database, user, password);
}

break;


// User name
case 'u':
case 'U':
{
	printf("=Enter new user name:\n");
	scanf("%s", user);
	getchar(); // Swallow Enter
	printf("Host=%s, Database=%s, User=%s, Password=%s\n",
		host, database, user, password);
}

break;

// Password
case 'p':
case 'P':
{
	printf("=Enter new password:\n");
	scanf("%s", password);
	getchar(); // Swallow Enter
	printf("Host=%s, Database=%s, User=%s, Password=%s\n",
		host, database, user, password);
}
break;

// Enter
case '\n':
{
	#if(0) // For testing
	printf("Enter\n\n");
	#endif

	// Set done flag
	done=1;
}
break;

// All done
case 'x':
{
	#if(1) // For testing
	printf("=All done\n");
```

```
                                #endif

                                // Set done flags
                                done=1;
                                alldone=1;
                        }
                }

                //. Values accepted
                if(done)
                {
                        // Reset done flag
                        done=0;

                        // Done with these parameters
                        break;
                }
        }
    }
}
#endif

#if(1)
// Function to read one item from simple ini file
// with lines of the form key=value
// without sections (no [section]s)
// and filename INIFILE
// Open ini file for reading
// Read line with fgets()
// Parse line for key using strtok()
// If key found, parse line for value using strtok()
// Process value for the key (e.g., set a variable named key to value)
// Close file
// Repeat by calling function for each key value wanted

int ReadIniValue(char *key, char *value)
{
        // Local variables
        char filename[BUFFER_SIZE+1]; // Ini filename
        static int file_open;
        static FILE *fptr=0; // File pointer

        // Assign filename
        strcpy(filename, INIFILE);

        // Open ini file
        if(!file_open)
        {
                fptr=fopen(filename, "r");
        }

        // Close ini file if NULL key
        if(key==NULL)
        {
                // Close file
                fclose(fptr);
                if(VERBOSE)
                {
                        printf("Ini file closed\n\n");
                }
                return 0;
        }

        if((fptr==NULL))
        {
                // Error opening file
```

```
		printf("Unable to open ini file %s\n", filename);

		// Return error code
		return 1;
	}
	else // File opened OK
	{
		// Was file opened just now?
		if(!file_open)
		{
			// Show file opened message
			if(VERBOSE)
			{
				printf("Ini file %s opened\n", filename);
			}

			// Set file open flag
			file_open=1;
		}
	}

	// Scan line of file
	// Put value of key into value
	// printf("Key=%s, ", key);
	// printf("Length=%d\n", strlen(key));
	fscanf(fptr, "%s=", value);
	// printf("Value=%s\n", &value[strlen(key)+1]);

	// Return success
	return 0;
}
#endif

#if(1)
// Function to get measurement parameters from ini file
int GetParams(char *description, char *filename, int *sig_id, int *ref_id, int *sn,
	double *tau, int *af, double *frequency, char *host, char *database, char *user, char
*password)
{
	// Local variables
	char key[BUFFER_SIZE+1]; // Ini file key string
	char value[BUFFER_SIZE+1]; // Ini file value string

	// Get default parameters from ini file
	// This is done for all parameters in sequence
	// We do NOT update the ini file values
	// They have to be changed by manually editing the ini file
	// Ini file format is key=value, no [sections]
	// Strings are not quoted, no spaces (use _ instead)
	// All 12 items must be present, in correct order

	// Line 1 - Description
	strcpy(key, "description");
	ReadIniValue(key, value);
	if(VERBOSE)
	{
		printf("%s=%s\n", key, &value[strlen(key)+1]);
	}
	strcpy(description, &value[strlen(key)+1]);

	// Line 2 - Filename
	strcpy(key, "filename");
	ReadIniValue(key, value);
	if(VERBOSE)
	{
		printf("%s=%s\n", key, &value[strlen(key)+1]);
```

```
}
strcpy(filename, &value[strlen(key)+1]);

// Line 3 - Signal ID
strcpy(key, "sig_id");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
*sig_id=atoi(&value[strlen(key)+1]);

// Line 4 - Reference ID
strcpy(key, "ref_id");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
*ref_id=atoi(&value[strlen(key)+1]);

// Line 5 - TICC Module S/N
strcpy(key, "sn");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
*sn=atoi(&value[strlen(key)+1]);

// Line 6 - Tau
strcpy(key, "tau");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
*tau=atof(&value[strlen(key)+1]);

// Line 7 - Averaging Factor
strcpy(key, "af");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
*af=atoi(&value[strlen(key)+1]);

// Line 8 - Frequency
strcpy(key, "frequency");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
*frequency=atof(&value[strlen(key)+1]);

// Line 9 - Host
strcpy(key, "host");
ReadIniValue(key, value);
if(VERBOSE)
{
	printf("%s=%s\n", key, &value[strlen(key)+1]);
}
strcpy(host, &value[strlen(key)+1]);
```

```
	// Line 10 - Database
	strcpy(key, "database");
	ReadIniValue(key, value);
	if(VERBOSE)
	{
		printf("%s=%s\n", key, &value[strlen(key)+1]);
	}
	strcpy(database, &value[strlen(key)+1]);

	// Line 11 - User
	strcpy(key, "user");
	ReadIniValue(key, value);
	if(VERBOSE)
	{
		printf("%s=%s\n", key, &value[strlen(key)+1]);
	}
	strcpy(user, &value[strlen(key)+1]);

	// Line 12 - Password
	strcpy(key, "password");
	ReadIniValue(key, value);
	if(VERBOSE)
	{
		printf("%s=%s\n", key, &value[strlen(key)+1]);
	}
	strcpy(password, &value[strlen(key)+1]);

	// Done - Close ini file
	ReadIniValue(NULL, value);

	return 0;
}
#endif
```

Appendix II – TestDB Makefile

```
# Makefile for TestDB project

TestDB: TestDB.o -lpq
	gcc TestDB.o -lpq -o TestDB

TestDB.o: TestDB.c
	gcc -c TestDB.c
```

Appendix III – TestDB.ini Configuration File

```
description=TAPR_TICC_Clock_Data
filename=/home/bill/Data/TICC.dat
sig_id=2
ref_id=1
sn=1003
tau=1.0
af=1
frequency=10e6
host=192.168.2.40
database=ppd
user=postgres
password=root
```

Appendix IV – TestDB License