B Software Documentation

The software in the EMFF system can be initially grouped by its physical location: avionics software on the avionics Tattletale, metrology software on the metrology Tattletale, and ground station software on the EMFF laptop. The avionics Tattletale (TT8) houses software for avionics, communication, and control; the ground station and metrology TT8 computers each hold the majority of their own subsystems' software. However, in this appendix all software (SW) will be grouped by function, or subsystem: "avionics" (B.1) covers all SW on the avionics TT8 but does not address communication or the control test module in detail; "communications" SW (B.2) includes the communications modules on the avionics Tattletale and all ground station SW; "metrology" includes all SW on the metrology TT8; "control" (B.4) includes specific information on translating control test algorithms to C source code and integrating them into the avionics software.

B.1 Avionics Software and Operating System Overview – MAS (p.1-15)

"Avionics software" is the name for the software package loaded to the avionics Tattletale computer. The avionics software is composed of three parts: the initialization section, which establishes vehicle- or system-wide standards (e.g. number of vehicles), mathematical constants, hardware I/O interfaces, and global variables; the *main()* function, which contains the EMFF operating system (OS); and the many separate functions, or modules, which accomplish specific tasks relating to timing, control, communication, metrology, or computation.

B.1.1 Initialization

B.1.1.1 Version documentation

The history of the code should be documented in the top section. *CODE_VER* is a string established here, available at any point in the code for printout during debugging. Under this are notes regarding sections of the avionics code currently undergoing revision.

The version of code to which this document refers is "IT-2A test 3" – or version 3 of Integrated Test 2A.

B.1.1.2 Constants

B.1.1.2.1 System Constants and Data Sizes

These constants are important at the entire system level. They set the identification of the local vehicle, the number of vehicles in the system, number of stages in the operating system cycle, buffer sizes for incoming communications and metrology data, and sizes of data-holding vectors (e.g. PVA, MSA). Table B-1 lists system and data size constants and where they are used.

Table B-1: Avionics System and Data Size Constants

Identifier	*Name/Description*	*Used in function*	*Default value?*
VEH_ID	Vehicle identification # (0, 1, or 2)	MSA, comm code	0 = master
NUMBER_OF_VEHICLES	Number of vehicles in system	MSA, comm code	2 = test 2X
TSBUFSIZE	Buffer size for TPU channel (input from Metrology TT8)	main(): Comm and Met channel initial'n	256 (B)
SBUFSIZE	Buffer size for serial channel (comm hardware = DR2000)	main(): comm channel initialization (only)	256 (B)
NUM_OF_STAGES	Number of stages in the main OS loop	MainTimingInt()	5 for 2-vehicle ops
IR_Period	Sets control pd for *simulated* IR beacon	SetupIRInt()	350 (ms)
METRO_RAW_DATA_LENGTH	Storage buffer for incoming met data	GetMetroData()	100 (B)
NUM_MET_DATA_ELEMENTS	Sets vector length for collected met data	GetMetroData()	12 (elements)
NUM_PVA_ELEMENTS	Sets vector length for PVA	CreateLocal PVA()	9 (elements)
NUM_MSA_ELEMENTS	Sets vector length for MSA	CreateMSA()	13 (elements)
RW_PULSE_WIDTH	Sets PWM signal width for RW actuation	ControlTestcase1(), SmallControlInterrupt()	0.010 (ms)
C_PULSE_WIDTH	Sets PWM signal width for coil actuation	ControlTestcase1(), SmallControlInterrupt()	1.0 (ms)

B.1.1.2.2 Sensor and Actuator Calibration Constants

The second set of constants pertains to the hardware attached to the avionics TT8. There are hardware-specific calibration constants supplied by the manufacturer for the gyroscope, as well as variables reserved for calibrating input data from avionics sensors (gyro, tachometer) and scaling output data to actuators (reaction wheel, EM coils). There is also a 10^x scalar in this section – *x_INTEGER_SCALAR*, which is only used to preserve significant figures when converting data to *integer* variable type. Table B.1.2 lists the sensor and actuator calibration constants and where they are used.

Table B-2: Sensor and Actuator Calibration Constants

Identifier	*Name/Description*	*Used in function*	*Default value?*
GYRO_CAL	Gyro-specific calibration value	SmallControlInterrupt()	2050 (SN33239), 1987 (SN32846)
GYRO_MAX	A/D input calibration value	SmallControlInterrupt()	4095 (mV)
GYRO_MIN	A/D input calibration value	SmallControlInterrupt()	0 (mV)
GYRO_ZERO_MAX	Gyro-specific calibration value	SmallControlInterrupt()	2055 / 1990
GYRO_ZERO_MIN	Gyro-specific calibration value	SmallControlInterrupt()	2047 / 1980
COIL_ONE_SENSOR_ZERO	Coil current sensor cal - shift	SmallControlInterrupt()	0.0 (Amps, placeholder)
COIL_TWO_SENSOR_ZERO	Coil current sensor cal - shift	SmallControlInterrupt()	0.0 (Amps, placeholder)
COIL_ONE_ACTUATOR_SCALAR	Coil current calibration - scale	SmallControlInterrupt()	0.625 (mu → Amps, arbitrary)
COIL_TWO_ACTUATOR_SCALAR	Coil current calibration - scale	SmallControlInterrupt()	0.625 (mu → Amps, arbitrary)
COIL_ONE_ACTUATOR_ZERO	Coil current calibration - shift	SmallControlInterrupt()	50.0 (Amps, central PWM value)
COIL_TWO_ACTUATOR_ZERO	Coil current calibration - shift	SmallControlInterrupt()	50.0 (Amps, central PWM value)
RW_ACTUATOR_SCALAR	RWA current calibration - scale	SmallControlInterrupt()	20 (no units, 2.6315789 saved?)
RW_ACTUATOR_ZERO	RWA current calibration - shift	SmallControlInterrupt()	50.0 (Amps)
COIL_INTEGER_SCALAR	Multiplier converting float to int	CreateLocalPVA()	100.0 (dimensionless)
GYRO_INTEGER_SCALAR	Multiplier converting float to int	CreateLocalPVA()	10000.0 if rad; 100.0 if deg (d'less)

B.1.1.3 Included Files

The EMFF uses Tattletale Model 8 (TT8) with C language capabilities. The avionics code must include certain standard and specialized C source files to run on the TT8. Table B.1.3 lists the standard and Tattletale includes and where they can be found in Tattletale reference documents.

Table B-3: Included Files

File name	*Library Type*
stdio.h	Standard C
Math.h	Standard C
string.h	Standard C
tt8.h	TattleTale Model 8
tt8lib.h	TattleTale Model 8
tpu332.h	TattleTale Model 8
dio332.h	TattleTale Model 8
userio.h	TattleTale Model 8
stdlib.h	TattleTale Model 8
Time.h	TattleTale Model 8

B.1.1.4 Input and Output Channels

The input and output (I/O) pins on the TT8 are explained more fully in the avionics hardware section of this document <section 7>. However, the code labels for all I/O pins are defined at the top of the avionics code. Table B.1.4 lists the EMFF TT8 I/O pins with their variable designation and use.

Table B-4: I/O Channels

Identifier	*Name/description*	*TT8 Pin Type and #*
MET_CHAN	Metrology (Tx then Rx)	TPU 0
TPU_IR	IR beacon timing interrupt count/detection	TPU 1
TPU_TACH_DIR	Tachometer direction in	TPU 2
TPU_TACH	Tachometer data in	TPU 3
C2_CHAN	Coil 2 PWM signal out	TPU 4
RW_CHAN	RW PWM signal out	TPU 5
TPU_ML	Main Loop timing interrupt count/detection	TPU 6
TPU_ML2	PWM timing signal out	TPU 7
C1_CHAN	Coil 1 PWM signal out	TPU 8
COMM_ICHAN	Comm (receive)	Serial 14
COMM_OCHAN	Comm (transmit)	Serial 13
ADCHAN_GYRO	Gyro data in (OBSOLETE?)	(previously TPU 0)
ACTUATOR_STOP	OBSOLETE	(previously TPU 8)

B.1.1.5 Function Prototypes

Each specific task of the avionics software is accomplished in a separate function, or module. Table B.1.5 lists all function names and the purpose of each. For a complete treatment of the avionics software modules, see section B.1.3.

Table B-5: Function Prototypes (Avionics Software Modules)

Function name	*Description*
void SmallControlInterrupt(void)	Adjusts attitude control with local feedback between metrology data arrivals
void ControlTestcase1(float MSA[ ])	Executes major control algorithm
void SetupMainTimingInt(void)	Sets up main loop interrupt (channel and recognition)

void MainTimingInt(void)	Handler for main timing interrupt; changes to next stage within OS cycle
void SetupPWM(int TPU_Chan)	Sets up a PWM signal on channel TPU_Chan
void DefinePWM(int TPU_Chan, float P_width, float P_duty)	Changes attributes of an existing PWM signal
void SetupIRInt(void)	Sets up IR beacon interrupt (channel and recognition)
void IRInterrupt()	Handler for IR beacon interrupt; restarts OS cycle
void GetMetroData(unsigned short int metrology_data[])	Brings in new data from metrology TT8
void ClearString(char data[])	Empties a vector of characters
void GetT(char cs[])	General; retrieves a byte over TPU channel
void GetH(char data[])	General; retrieves byte over serial channel
void SendH(char cs[])	General; sends a byte over serial channel
void CreateLocalPVA(unsigned short int PVA_Local[], unsigned short int metrology_data[])	Pulls together local metrology and gyro/coil data into Primary Vehicle Array (PVA)
void Transmit_PVA(unsigned short int PVA_Local[], int PacketNumber, unsigned short int PVA_Local_Packaged)	Packages local PVA for transmission and sends over RF channel through DR2000
void Fetch_PVA(unsigned short int PVA_Remote[], unsigned short int PVA_Remote_Packaged)	Brings in remote PVA from RF channel through DR2000 and unpackages
void CreateMSA(float MSA[], unsigned short PVA_Local[],unsigned short PVA_Remote)	Combines all PVA data into MSA
void FloatToInt(void)	Casts a *float* as an *int*, saving X significant figures
int BitwiseFun(void)	Bitwise arithmetic needed to cast *float* as *int*
void ConvertMSA(float MSA[], int result)	Converts the real-number MSA to int form for transmit
void InitTACH(void)	Initializes tachometer
float GetTACH(void)	Polls tachometer for data

B.1.1.6 Global Variables

EMFF avionics code does use global variables. Global variables are unavoidable when programming the TT8 using interrupts (see section B.1.2 for explanation). Table B.1.6 lists and identifies the avionics global variables and notes the functions that use each. In the avionics software, global variables are grouped according to the functions that use them. NOTE: "unsigned short" implies type *int*.

Table B-6: Global Variables

Type	*Identifier*	*Name/description*	*Used in function(s)*	*Initial value?*
int	flag_cmd	N/A		1
int	testbit	N/A		0
int	loop_counter	**index** for OS stage count	main, MainTimingInt	0
int	loop_stage	holds actual OS stage	main, MainTimingInt	0
int	loop_order [NUM_OF_STAGES]	order/numbers of OS stages	MainTimingInt	{1,2,3,4,5}
int	stage_length [NUM_OF_STAGES]	vector with MainTimingInt timeout for each OS stage	main, MainTimingInt	(100,200,300, 400,500)
int	int_flag	interrupt flag	main, MainTimingInt	1
int	inner_loop_counter	gyro calibration counter	main	0
int	gyro_new_cal	gyro calibration value	main, SmallControlInterrupt	0
ExcCFrame	framebuf0 framebuf1 framebuf2	standard buffers	InstallHandler(…) [standard]	
int	Packet_Number	packet ID number	Transmit_PVA, Fetch_PVA	48 (ASCII A)
unsigned short	PVA_Local [NUM_PVA_ELEMENTS]	Primary Vehicle Array vector (for comm transmission)	CreateLocalPVA, CreateMSA, Transmit_PVA, Fetch_PVA	N/A
unsigned short	PVA_Remote [NUM_PVA_ELEMENTS]	Primary Vehicle Array vector (for comm transmission)	CreateLocalPVA, CreateMSA, Transmit_PVA, Fetch_PVA	N/A
unsigned	PVA_Local_Packaged	PVA_Local formatted into	Transmit_PVA, Fetch_PVA	N/A

short	[NUM_PVA_ELEMENTS*2]	bytes for comm trans		
unsigned short	PVA_Remote_Packaged [NUM_PVA_ELEMENTS*2]	PVA_Remote formatted into bytes for comm trans	Transmit_PVA, Fetch_PVA	N/A
float	c_rw_voltage	commanded RWA voltage	SmallControlInterrupt, ControlTestcase1	0.0
float	cmd_PWM	RW cmmanded PWM signal	SmallControlInterrupt, ControlTestcase1	N/A
float	c_current1	commanded current coil 1	SmallControlInterrupt, ControlTestcase1	0.0
float	c_current2	commanded current coil 2	SmallControlInterrupt, ControlTestcase1	0.0
float	a_current1	actual current reading coil 1	SmallControlInterrupt, ControlTestcase1	0.0
float	a_current2	actual current reading coil 2	SmallControlInterrupt, ControlTestcase1	0.0
float	c_torque	commanded torque, rad/s	SmallControlInterrupt	0
float	theta_int	theta … radians	SmallControlInterrupt	0.0
float	theta	angle – radians	SmallControlInterrupt	0.0
float(?)	t_now	formerly ulong	SmallControlInterrupt	0
float(?)	t_last	formerly ulong	SmallControlInterrupt	0
float(?)	dt	formerly ulong	SmallControlInterrupt	0
float	thetadot	angular rate - rad/s	SmallControlInterrupt	0
float	desired_theta_now	desired angle, Test 1B only	SmallControlInterrupt	0
float	desired_theta_last	desired angle, Test 1B only	SmallControlInterrupt	0
float	old_tach	smooth out tach data	SmallControlInterrupt	0
int	all_stop	= 1 indicates all actuators off	OBSOLETE	0
float	num	holds element of MSA	CreateMSA (?)	N/A
int	sign	neg = 1; pos = 0	FloatToInt	N/A
int	mantissa	base, float number	FloatToInt	N/A
int	exponent	exponent, float number	FloatToInt	N/A

* the “unsigned short int” variable type is necessary for any variable which will be transmitted over the RF channel – that is, which is handled by the comm subsystem. Therefore the PVA data must be formatted such that it can be expressed in type *int* without losing any significant digits. See B.1.3.14, `Create_Local_PVA()` for more information on this typecasting and formatting.

B.1.2 Main function and Operating System (OS)

B.1.2.1 Background and Versions

The operating system (OS) of the EMFF avionics went through several iterations before it was actually written. We tried to design a simple system with loops and a linear progression, but eventually learned that the tasks the OS had to accomplish were too complex to handle without using interrupts. We also looked at event-based interrupts, which we felt were more appropriate than timing-based interrupts, but found this to be too flaky when used on its own. So the OS that was finally coded, in January 2003, is both timed and event-driven – the driving event is the IR beacon from the metrology system. However, when testing before integration or without metrology, there is another time-based interrupt set to simulate the IR. This is explained in greater detail in the following section (B.1.1.2, OS Interrupt Timing Cycle). For a top-level diagram of the avionics software task progression, see Figure B-1: Overall Avionics Software Flowchart.

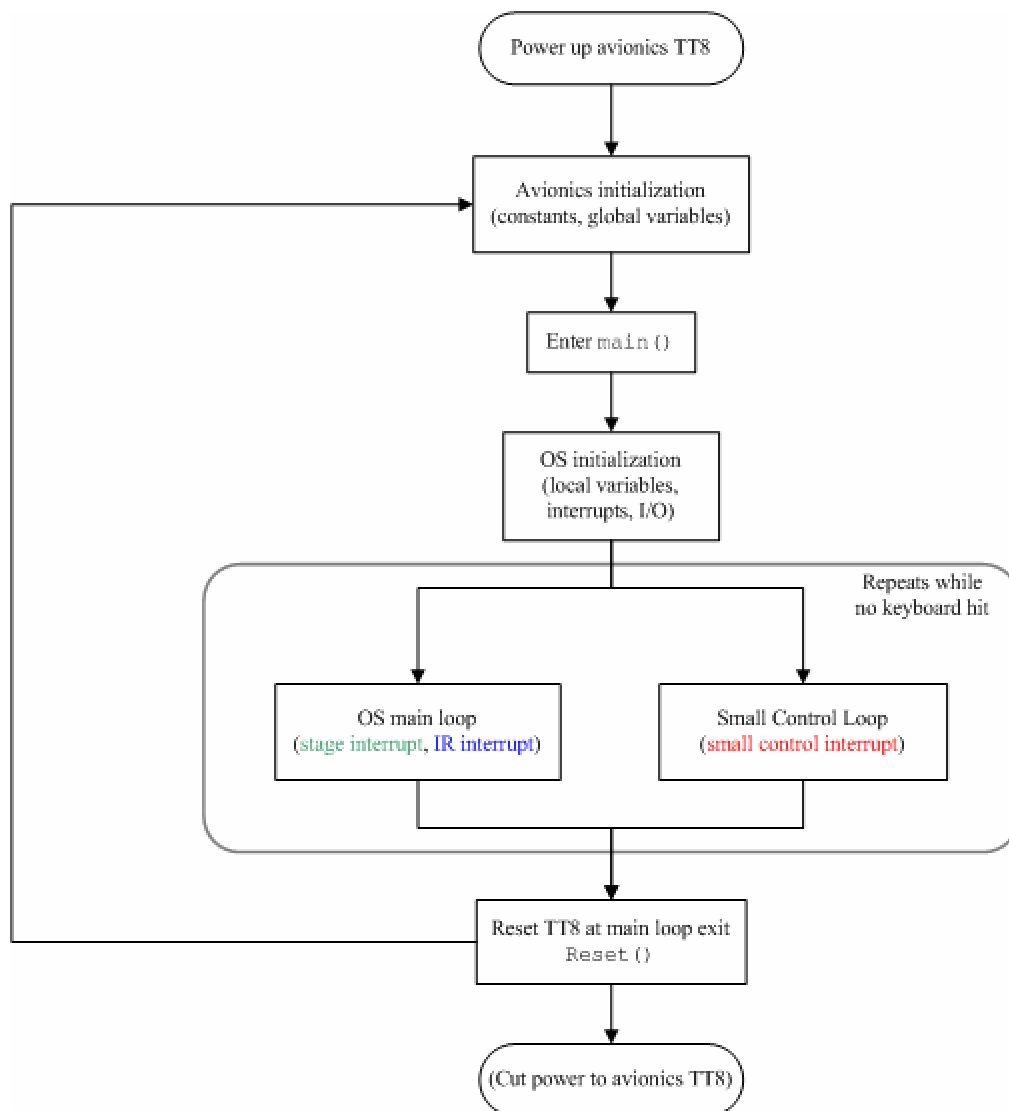


Figure B-1: Overall Avionics Software Flowchart

The version of code documented in this Appendix is "IT-2A Version 3" – used throughout April and May 2003 on the EMFF vehicle(s). There is some documentation recording the evolution of the code through this version at the top of the C source file. Later versions (as of May 2003 there are "4" and "5") begin the reorganization of the main code – including simplifying the main thread and deleting extraneous sections and commands as well as streamlining modules. The plan is for versions 4 and 5 of IT-2A to be incorporated into future tests (e.g. IT-2B, 2C, 2D).

The specific file to which all examples are referenced in this document is as follows:
Metrowerks CodeWarrior Project "**Main Project Codev5 2A with MSA test.mcp**"
… linked at this time to
"**Main_Code_v3_IT-2A_29APR2003_TEST_modified_for_if_metro_doesnt_work.c**"
(May 1, 2003. no explanation why the Project version says 5 while the code version is still 3.)

B.1.2.2 OS Interrupt Timing Cycle (overview and design)

B.1.2.2.1 Tasks and Limiting Factors

The high-level tasks of the operating system are the following:

1. Collect data on vehicle position/motion (local data) ("Primary Vehicle Array" or PVA)
2. Communicate local data to other vehicles and gather remote data from other vehicles
3. Combine all data into Master State Array (MSA)
4. Run control algorithm on MSA data
5. Implement output from control algorithm by commanding actuators (coils and reaction wheel)

All of the above tasks must happen in a single cycle of the OS. There are two main limiting factors on length of the OS cycle: the frequency of updates required to run the control algorithm successfully (high end) and the rate at which metrology can supply new data (low end). The speed of the computer is not a limiting factor; when using interrupts, the CPU and TPU capacity is more than sufficient to run at the speed required by control.

The speed of the communication subsystem (transmission over the RF channel among multiple vehicles and a ground station) is also a factor, but the communication subsystem is not fully implemented in the code documented here, so its limits are not yet certain. For more on comm, see section 2 in this Appendix.

B.1.2.2.2 Stage Progression

The avionics software main function is broken up into stages. Theoretically, each high-level task in the OS is addressed in one stage of the function. The time allotted to complete each stage is determined by an "inner loop" interrupt. The timing of the interrupt is determined by the global integer variable *stage_length[NUM_OF_STAGES]* – defined at the top of the code as a vector containing the length of each OS stage in milliseconds. If any one high-level task times out, the OS proceeds to the next stage. For example, if there is an error bringing in metrology data, rather than waste time (causing further drift) the OS will proceed through the stages anyway, and extrapolate the control input at the final stage using metrology data from the previous cycle.

B.1.2.2.3 Infrared Interrupt

To optimize accuracy, the five stages of the OS should run every time fresh local data is available from the metrology subsystem. Therefore the system timing is driven by an interrupt based on the infrared (IR) beacon flash from the metrology Tattletale (TT8), which signals a new metrology data-gathering cycle. The input pin through which avionics TT8 receives the IR signal from metrology TT8 is TPU channel 1.

B.1.2.2.4 Simulated IR

The avionics TT8 is also set up to run a simulated IR interrupt when the metrology subsystem is not available or when debugging the avionics subsystem in an isolated environment. The standard simulated IR cycle length is defined to be 350ms (*#define IR_Period*, see Table 1) but may be adjusted according to need while testing.

B.1.2.2.5 Intermediate Adjustments – Small Control Loop

Since the main loop is set to run on the order of <5Hz based on the metrology subsystem, there is a substantial amount of time for drift to occur between arrivals of fresh metrology data. To ensure that the system does not drift beyond the point of recovery, more frequent adjustments are made to local vehicle control using measurements taken from the gyroscope on board each vehicle. This "Small Control Loop" (SCL) is run on a timed interrupt approximately every 100 ms. For sequence and details of the SCL, refer to section 1.3 (Modules), under `SmallControlInterrupt()`.

B.1.2.2.6 Summary of Timing Design

The process developed in January 2003 approximately follows the overall timing cycle shown in Figure B-2.

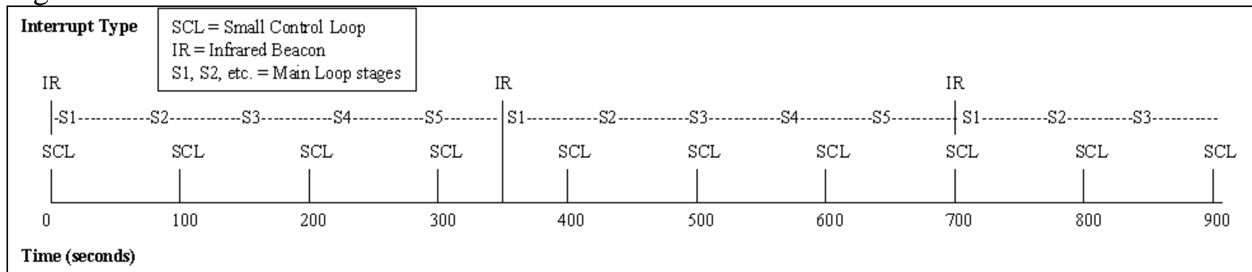


Figure B-2: Operating System Timing Cycle

For a short time at each SCL interrupt the processor executes attitude adjustments using the gyro data. metrology data collection, RF communication, and main control execution all occur at each IR interrupt. NOTE: In reality the period on the SCL is not exactly 100ms; therefore there is no simultaneous interrupt at multiples of 700ms.

B.1.2.3 OS Implementation: `main()` Function and Interrupts

When the avionics TT8 is powered up, it runs through the avionics initialization code (B.1.1) and then enters the OS proper in the `main()` function. (see Figure B-1 in section B.1.2.1). The `main()` function contains initialization of all interrupts (IR, main timing, and small control loop), hardware, pulse width modulated (PWM) signals, and data arrays.

B.1.2.3.1 OS Initialization

Figure B-3 follows the thread of operations in the `main()` function through this initialization process and the entrance to the OS loop.

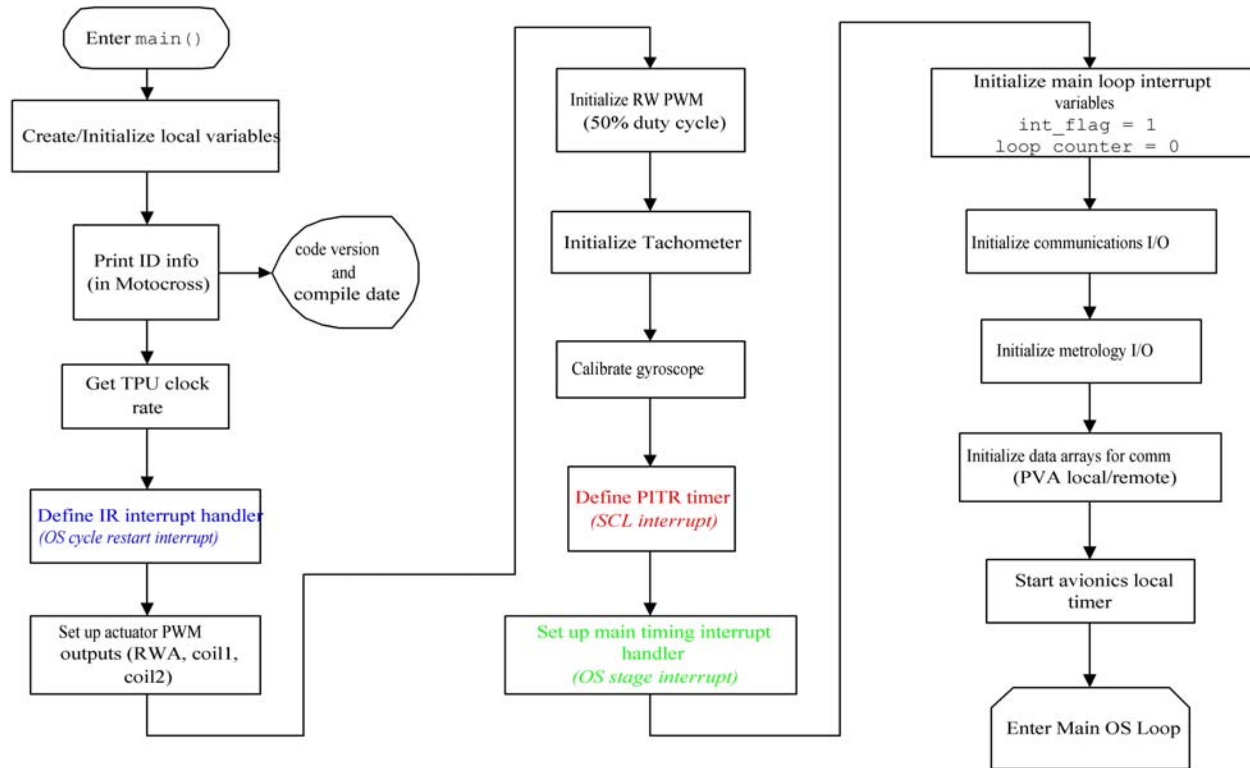


Figure B-3: OS Initialization Sequence

Create and Initialize Local Variables

Aside from the global variables noted in section B.1.1.6 (Table B-6), the avionics code utilizes many variables local to the main function. Table B-7 lists the local `main()` function variables and their significance.

Table B-7: Local Variables, main() Function

*Type**	*Identifier*	*Name/description*	*Initial value?*
`float`	`cal_loop`	counter for gyro calibration loop	2
`int`	`temp_cal_var`	used in gyro calibration	-
`int`	`i`	loop counter – PVA debug	0
`unsigned char`	`done`	exit condition for main loop	0
`char`	`data[257]`	debug for comm – OBSOLETE	`NULL`
`int`	`met_stillopen`	test MET_CHAN close Tx - before reopen Rx	1
`unsigned short int`	`metrology_data [NUM_MET_DATA_ELEMENTS]`	vector to hold incoming metrology data	`{0}`
`unsigned short int`	`PVA_Local [NUM_PVA_ELEMENTS]`	holds PVA data for local vehicle (1, or A)	`{0}`
`unsigned short int`	`PVA_Remote [NUM_PVA_ELEMENTS]`	holds PVA data for remote vehicle (2, or B)	`{0}`
`unsigned short int`	`PVA_C[NUM_PVA_ELEMENTS]`	holds PVA data for third vehicle - FUTURE	`{0}`
`float`	`MSA[NUM_MSA_ELEMENTS]`	holds MSA information for ControlTestcase1	`{0}`
`int`	`result`	called by ConvertMSA(MSA, result)	0
`long`	`sammy`	holds TPU TCR 1 clock value; type long required by TT8 function TPUGetTCR1()	-
`int`	`result1, result2`	comm initialization – channel open success	-

* the "*unsigned short int*" variable type is necessary for any variable which will be transmitted over the RF channel – that is, which is handled by the comm subsystem. Therefore the PVA data must be formatted such that it can be expressed in type *int* without losing any significant digits. See B.1.3.14 (`Create_Local_PVA()`) for more information on this typecasting and formatting.

Print ID info

When running the avionics TT8 from Motocross (loading program), it will print identification information at the beginning of every run. It prints the title of the code (not filename), version number as defined in the label at the top of the file, and the time/date of the last compilation.

TPU clock

The temporary variable "*sammy*" holds the value returned here from `TPUGetTCR1()`. More information on this function can be found in the *Tattletale Model 8 manual, ANSI-C version*, found online as "TT8C_Man.pdf" in \\aero-astro\CDIO3\07. Reference\4. Tattletale Manual\ .

IR Interrupt Handler

Defining the infrared interrupt includes three function calls: setting up the interrupt, installing the interrupt handler, and enabling the interrupt. Details about how to use each of these functions can be found in the text(s) referenced for each function.

- `SetupIRInt()`
  Reference: section B.1.3.6 (`SetupIRInt`)
  NOTE: The period of the *simulated* IR interrupt (PWM count) is set in this function.
- `InstallHandler(IRInterrupt, TPU_INT_VECTOR + TPU_IR, &framebuf2)`
  References: TT8C_Man.pdf, section 5 page 17 (`InstallHandler`)
  section B.1.3.7 (`IRInterrupt`)
- `TPUInterruptEnable(TPU_IR)`
  Reference: TT8C_Man.pdf, section 5 page 42

Set up actuator PWM outputs

The system actuators (reaction wheel and two superconducting coils) are commanded by the control algorithm using a pulse-width modulated (PWM) signal – which is put out through the avionics TT8. Refer to section 1.3.x (`SetupPWM`) for more detailed information. The PWM signal itself is monitored by the TT8 TPU – for more information on this process, see the specific TPU reference manual "tpupn17 – PWM Function.pdf" in
\\aero-astro\CDIO3\07. Reference\4. Tattletale Manual\App Notes\ .

Initialize RW PWM

The reaction wheel PWM signal is specifically initialized to a 50% duty cycle immediately after the channel is set up. This is the "zero" setting for the reaction wheel. It is this opinion of this EMFFORCER that there should also be initialization commands to re-define the PWM for each coil – and that since we had not yet powered up the coils at this version of the test we had not included those commands. However, please note that this is speculation.

Initialize Tachometer

Simply calls the `InitTACH()` function. For more information refer to B.1.3.x (`InitTACH`).

Calibrate gyroscope

The gyroscope is calibrated by running a 90-rep loop that incorporates the new value into the old values. It changes the global variable `gyro_new_cal` using the local variables `temp_cal_var` and `cal_loop`. For details on the function used to access the gyro reading), refer to TT8C_Man.pdf, section 5, page 10 (`AtoDReadMilliVolts`).

Define PITR timer (Small Control Loop)

The Small Control Loop (SCL) runs on an interrupt different from that which governs the Main Timing and Infrared interrupts. The periodic interrupt timer (PITR) is an ultra-fast timer available on the TT8 which runs separately from the main thread. Since the SCL must interrupt with the highest frequency, it is set using this timer. The PITR is mentioned on pages 6-20 and 5-17 in TT8C_Man.pdf.

Set Up Main Timing Interrupt

The Main Loop (ML) Timing Interrupt governs the length of each stage of the OS. Functions used to set up the
- `SetupMainTimingInt()`

    Reference: section B.1.3.x (`SetupMainTimingInt`)

    NOTE: The *counter* PWM signal is started in this function. We set a signal to fire every millisecond through pin *TPU_ML2*, and tie that pin to *TPU_ML* – through which we count the pulses, thereby incrementing the timer for the Main Timing and *simulated* IR interrupts.
- `InstallHandler(MainTimingInt, TPU_INT_VECTOR + TPU_ML, &framebuf1)`

    References: TT8C_Man.pdf, section 5 page 17 (`InstallHandler`)
    section B.1.3.x (`MainTimingInt`)

NOTE: This set-up sequence does *not* call "`TPUInterruptEnable(TPU_ML)`" – because that function is called already in `MainTimingInt()`. The interrupt timeout length (i.e. number of pulses counted before *int_flag* is set) on *TPU_ML* is set differently depending on which stage the OS is in – governed by the global variable *stage_length(5)*.

Initialize Main Timing Interrupt

After setting up the main timing interrupt, it is necessary to begin the sequence by running the handler: `MainTimingInt()`. This does the following:
- increments global variable *loop_counter* (from 0, as it was initialized, to 1)
- sets `MainTimingInt` timer to *stage_length[loop_counter]* (stage_length[1] = 100ms)
- sets *int_flag* = 1 (TRUE)

When `MainTimingInt()` returns, the remainder of the initialization process does the following:
- sets *int_flag* = 1 (TRUE) (again)
- resets *loop_counter* = 0 (for entrance into main loop?)

Initialize Communications I/O

Comm I/O initialization includes the following processes:
- Open transmit channel – `TSerOpen(COMM_OCHAN, …)` – with error message

Reference: TT8C_Man.pdf, section 5 page 44 (`TSerOpen`)
- Open receive channel - `TSerOpen(COMM_ICHAN, …)` – with error message
Reference: TT8C_Man.pdf, section 5 page 44 (`TSerOpen`)
- Provide buffer for incoming comm data (Serial input)
Reference: TT8C_Man.pdf, section 5 page 31 (`SerSetInBuf`)

Initialize Metrology I/O

Metrology I/O initialization includes the following processes:
- Open TT8-TT8 channel to *transmit* - `TSerOpen(MET_CHAN, …, OUTP, …)`
Reference: TT8C_Man.pdf, section 5 page 44 (`TSerOpen`)
- Send initializing "start byte" to metrology TT8 – "go" command (doesn't currently work)
Reference: TT8C_Man.pdf, section 5 page 45 (`TSerPutByte`)
- Close transmit channel – with error message – `TSerClose(MET_CHAN)`
Reference: TT8C_Man.pdf, section 5 page 43 (`TSerClose`)
- Open TT8-TT8 channel to *receive* - `TSerOpen(MET_CHAN, …, INP, …)`
Reference: TT8C_Man.pdf, section 5 page 44 (`TSerOpen`)
- Flush metrology input channel to clear out any junk accumulated on initialization
Reference: TT8C_Man.pdf, section 5 page 43 (`TSerInFlush`)

Initialize Data Arrays

This sequence ensures that the Primary Vehicle Array (PVA) data are set to 0 before they are used in the main loop. The arrays currently initialized here are those used by the comm sequence for vehicle 1 in a two-node network (what we have): *PVA_Local*, *PVA_Remote*, *PVA_Local_Packaged*, *PVA_Remote_Packaged*. The "–Packaged" arrays are the original arrays (unsigned short integer, 2 bytes per element) split into bytes for transmission to the comm hardware (DR2000) and over the RF channel.

Start Avionics Local Timer

The PVA data requires a local timestamp in order to link data for the local vehicle (*PVA_Local*) to the simultaneously collected set of data from other vehicle(s) (*PVA_Remote*). This is achieved through the TT8 embedded "stopwatch" function. For more information, refer to TT8C_Man.pdf, section 5 page 38 (`StopWatchStart()`).

Enter Main Loop

The next step is to enter the Main Loop – set apart by a "while" loop with exit condition of a keystroke. The main loop is explained in detail in the following section, B.1.2.

B.1.2.3.2 OS Main Loop and Interrupts

Figure B-4 traces the flow of the OS Main Loop *without* the Small Control Loop interrupts. The SCL runs separately on its PITR (periodic interrupt timer) at a higher frequency of interrupt. For more information on the SCL, see its module description (B.1.3.x, `SmallControlInterrupt`).

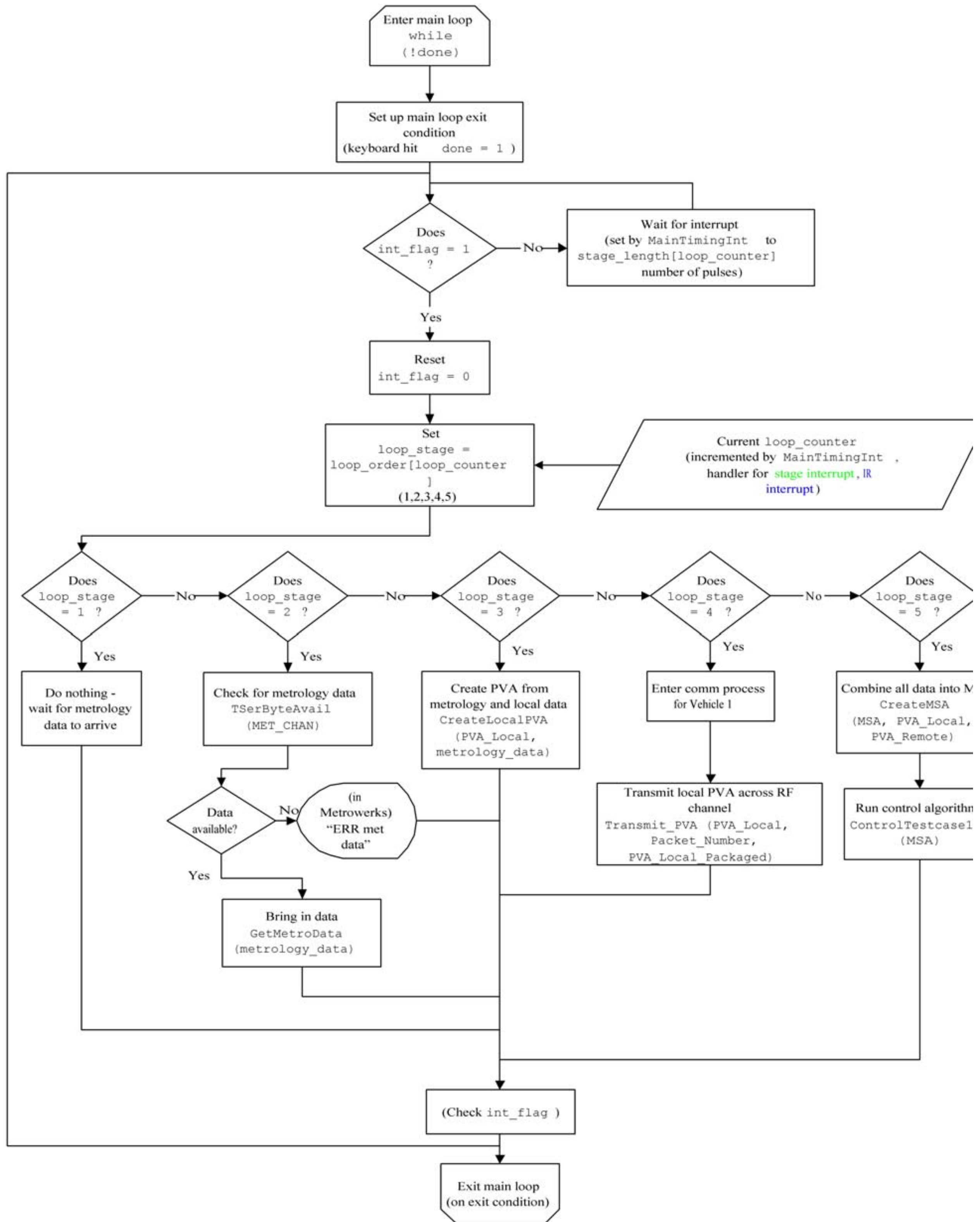


Figure 0-4: OS Main Loop Cycle

Enter Main Loop

The "Main Loop" cycle of the OS – containing the functions leading up to execution of the control algorithm – is set apart inside a simple "while" loop. Until the exit condition is met (*done* = 1), the OS continues to address interrupts and cycle through the stages.

Main Loop Exit Condition

The exit condition for the main loop is detection of a keyboard hit (reference TT8C_Man.pdf section 5 page 18, `kbhit()`) – but this can only register while testing the avionics code through a serial cable to a host PC running Motocross (the TT8 loading program). When testing the avionics code as burned to FLASH memory and disconnected from the PC, the only exit is to manually reset the TT8 by pressing its [orange] reset button or cutting power.

The Variable `int_flag`

The global variable *int_flag* (interrupt flag) triggers entry into the main() function switch/case statement, which contains the OS stages in sequence. *Int_flag* is…
- INITIALIZED to 1 at the top of the avionics code
- SET to 1 in `MainTimingInt`, which runs both whenever the main timer (on *TPU_ML*) interrupts, or when `IRInterrupt` runs (when there is an IR beacon flash, or the simulated-IR timer on *TPU_IR* interrupts; `IRInterrupt` calls `MainTimingInt`)
- CLEARED (set to 0) in the `main` function, immediately after recognition that it was set.

The Variables `Loop_Stage` and `Loop_counter`

The global variable *loop_stage* determines which task in the OS cycle is executed next. After clearing *int_flag*, the next action is to assign a value to *loop_stage* based on the current value of the variable *loop_counter*. The global variable *loop_counter* holds the *index* of the current OS stage. The range of this index is 0-4. *Loop_counter* is an index of two vectors:
- the current stage number (e.g. 3)
  - set of stage numbers is held in the global variable vector *loop_order*
  - range of stage numbers is 1 to 5
  - stage number = *loop_counter* + 1
- the length of time (e.g. 150) allotted for that stage (e.g. 3) before the main timer interrupts
  - set of time lengths is held in the global variable vector *stage_length*, in milliseconds

OS Stage 1 – Wait

In the case that the stage number is 1 (this means that *loop_counter* = 0), the only task of the OS is to wait until it receives metrology data. It appears that this stage is left over from a time when the OS cycle restart was to be triggered by the arrival of data from the metrology TT8. Regardless, the avionics system does nothing in this case and returns to the beginning of the main OS loop (while(!done)) to check *done* and *int_flag* again.
NOTE: If the main timer interrupts before the process is complete, an error message is printed to the screen (when working while connected to a PC running Motocross).

OS Stage 2 – Metrology

When the stage number is 2 (*loop_counter* = 1), the OS checks for metrology data. This calls the following steps:
- Check to see if the data arrived, i.e., if there is a data byte waiting on the metrology channel
  Reference: TT8C_Man.pdf, section 5 page 42 (`TSerByteAvail(MET_CHAN)`)
- If no data have arrived, print error to screen (only when connected to PC with Motocross)
- If data are there, bring in the data using `GetMetroData(metrology_data)`.
  References: section B.1.3.11 (`GetMetroData(metrology_data[])`)
  section B.1.3.12 (`GetT(data[])`)
  TT8C_Man.pdf, section 5 page 43 (`TSerGetByte()`)
- Return to beginning of main loop; check *done* and *int_flag*

NOTE: If the main timer interrupts before the process is complete, an error message is printed to the screen (when working while connected to a PC running Motocross).

OS Stage 3 – PVA

When the stage number is 3 (*loop_counter* = 2), the avionics TT8 must collect a local timestamp, local gyro and tach readings, and the metrology data collected in stage 2; then it must format this data into *unsigned short int* type and enter into the *PVA_Local* vector. The reason for the uniform typecasting is to prepare the data for transmission over the RF channel in the communication process.

Reference: section B.1.3.14 (`CreateLocalPVA(PVA_Local[])`)

After this, follow the same procedure: return to beginning of main loop; check *done* and *int_flag*
NOTE: If the main timer interrupts before the process is complete, an error message is printed to the screen (when working while connected to a PC running Motocross).

OS Stage 4 - Comm

When the stage number is 4 (*loop_counter* = 3), the avionics TT8 enters the communication process. For the two-node network (one vehicle + ground station (GS), RF transmission from vehicle to GS only), this is simple; there is
- one conditional statement that checks the vehicle ID (if *VEH_ID* == 0), and proceeds to…
- transmit or receive accordingly.

For a multi-vehicle network, the communication algorithm and code is substantially more complex; for more comm theory and design refer to section B.2 (Communication Software).
References: section B.1.3.17 (`Transmit_PVA(PVA_Local[], …)`)
After this, follow the same procedure: return to beginning of main loop; check *done* and *int_flag*
NOTE: If the main timer interrupts before the process is complete, an error message is printed to the screen (when working connected to a PC running Motocross).

OS Stage 5 - MSA and Control

When the stage number is 5 (*loop_counter* = 4), the avionics TT8 must run two related processes sequentially:
- create the Master State Array (MSA)
  Reference: sec B.1.3.19 (`CreateMSA(MSA[], PVA_Local[], PVA_Remote[])`)
- run the control test case using that MSA.
  Reference: Appendix A: Control

NOTE1: For an expanded system, the comm would handle and log an MSA transmission to the ground station (GS) as well as the PVA transmission sequence. To prepare the MSA for this

transmission, it must be typecast all as *unsigned short int* – which is accomplished through `ConvertMSA(MSA[], result)`. However, this code has not yet been tested on the floor.
NOTE2: If the main timer interrupts before the process is complete, an error message is printed to the screen (when working connected to a PC running Motocross).

B.1.3 Avionics Software: Modules

B.1.3.1 SmallControlInterrupt

NOTE: ALL I/O for the Small Control Interrupt (SCL) is GLOBAL because the function is an interrupt handler, which cannot be passed any inputs or return any values.
Global Variables - Inputs: determined in overall control calculations and only *referenced* here
    c_speed (commanded RW speed),
    c_current1, c_current2 (commanded current for coils)
Global Variables – Inputs/Outputs (modified):
    a_current1, a_current2 (actual current value in coil 1 or 2)
Other references:
    TT8 library function AtoDReadMillivolts(ADCHAN_x) (TT8C_Man.pdf section 5)
    EMFF function GetTACH() (section B.1.3.9)
    TT8 function StopWatchTime() (TT8C_Man.pdf section 5)
    Gyro calibration variables and procedures (refer to B.1.2, B.1.1)
    DefinePWM(*inputs*) (B.1.3.5)
    Defined channels for signal output to each actuator (RW, coil1 coil 2) (see B.1.1)
Notes:
    The Small Control Loop interrupts the OS system progress approximately every 100ms (based on the PITR timer set in the *main()* function) to collect data on the actuators and compensate for drift in control due to time lag. It…
    - gathers fresh gyroscope and tachometer data with the most recent (but older than gyro/tach) desired actuation levels commanded by control.
    - adjusts the PWM signal put out to the actuators, compensating for control/position drift as well as for weakening batteries powering the coils/RW.
    This function is extremely well commented. See source file
    (“Main_Code_v3_IT-2A_20APR2003_TEST_modified_for_if_metro_doesnt_work.c”) for a detailed walkthrough and step-by-step explanation of SCL calculations and commands.
    The StopWatchTime function returns the time in microseconds in variable type *ulong*, but we have changed it temporarily to type *float* for debugging
    The coil current sensors are not operational at print time for this document; code involving commanded and actual coil current has not been adequately (at all?) tested/debugged

B.1.3.2 ControlTestCase1 <BB>

The purpose of this section is to control the vehicle. This section is referred to as the big control loop because it is where the overall control is calculated. It differs from the small control loop because it updates slower and uses metrology position/attitude data in addition to local gyro and tachometer feedback data.

First this modules stores all the MSA data into an array. It then set up some counters, constants, and variables. The most important of these are control_output, input_state, and gains.

The array control_output is the output command of the controller. The array has three values for test case 2ab. The first value is the commanded magnetic moment, mu, for coil 0 -- the coil that starts parallel to the fixed coil, the big coil. The second value is the commanded magnetic moment for the other coil, and the last value is the commanded torque for the reaction wheel. The input_state array contains six values which are computed from the MSA values mentioned above. The first value is the x distance between the vehicles, the second value is the y distance, and the third value is theta. The last three values are the derivatives of the first three. For more detail on this see the modeling section of Appendix A. The gains matrix is a matrix that is the size of control_output by the size of input_state, in this case 3 by 6. These values are taken from the Control team's work and constitute the core of the controller for the system. For more information see Appendix A.

The next part of this module performs a matrix multiplication of the gains matrix and the input_state array. The results of this multiplication is stored in control_output. The values of control_output are then manipulated to provide commanded current to the coils -- c_current1 and c_current2 – and commanded voltage to the reaction wheel. The conversions for c_current1 and c_current2 are straightforward. The conversion for c_torque is a bit more tricky and is dealt with in Appendix A.

B.1.3.3 MainTimingInt

Inputs: none
Outputs/Return: none
Global variables accessed: loop_counter, int_flag, stage_length[]
Other references: TPU_ML (main loop timing/interrupt pin),
Processes:

Increment loop_counter
Clear the interrupt on TPU_ML
Set the number of counts TPU_ML should register before next interrupt
→ set number of counts = stage_length[loop_counter]
Initialize TPU_ML to begin counting again
Reset int_flag = 1
Conditional: if this process has run the last stage (loop_counter == NUM_OF_STAGES), then the main loop has completed a full cycle before registering an IR interrupt. At this point the TT8 should reset and begin again; instead for debugging purposes simply reset loop_counter to 0 (to restart the OS cycle).

Notes:

Called as handler for TPU_ML timeout interrupt; *also* called from IRInterrupt (handler for TPU_IR interrupt, which happens either from metrology IR or from simulated IR PWM).

The IR-simulation PWM signal is set up and defined at the beginning of the `MainTimingInt()` module; this code should be commented out when running the avionics with metrology connected.

B.1.3.4 SetupPWM

Inputs: *int* TPU_Chan
Outputs/Return: none
Global variables accessed: none
Other references: TPU function: PWM → "tpupn17 - PWM Function.pdf" in TT8 App Notes

Processes: Uses process as defined in Motorola spec sheets to set up a pulse width-modulated function going out on channel "TPU_Chan" with given period length and duty cycle.
Notes: Refer to TT8 app notes for more information on function setup and PRAM offsets.

B.1.3.5 DefinePWM

Inputs: *int* TPU_Chan, *float* P_width (milliseconds), *float* P_duty (percent 0→100)
Outputs/Return: none
Global variables accessed: none
Other references: TPU function: PWM → "tpupn17 - PWM Function.pdf" in TT8 App Notes
Processes:

Prints PWM change information to the screen (if hooked to PC running Motocross)
Conditional: IF NEW PWM DOWN TIME IS LESS THAN 10 ms
→ set new pulse width and percent duty using TPU PWM function
ELSE → print error message!

Notes: the error message from a down time of *greater* than 10 ms is to limit the *upper* end of the PWM signal. There is a maximum time that you can set the pulse width (0x8000 [hex]) - duty cycle combination, and the conditional checks that.

B.1.3.6 SetupIRInt

This function sets up the infrared interrupt receiver (NOT the interrupt handler!)
Inputs: none
Outputs: none
Global variables accessed: none
Other references:

TPU_IR (pin detecting incoming interrupt)
IR_Period (defined length for simulated IR pulse)
TPU function: ITC → "tpupn16 - ITC Function.pdf" in TT8 App Notes

Processes:

Uses process as defined in Motorola spec sheets to receive an incoming pulse and register as an interrupt. Basically: disable chan (TPU_IR) → set function parameters → enable chan.
Notes: Refer to TT8 app notes for more information on function setup and PRAM offsets.

B.1.3.7 IRInterrupt

This function is the handler for the incoming IR beacon interrupt.
Inputs: none
Outputs: none
Global variables accessed: loop_counter
Other references:

TPU_IR (pin detecting incoming IR or simulated IR interrupt)
TPU_ML (main loop timing/interrupt pin – timeout varies according to OS stage),
MainTimingInt(); (section B.1.3.3)
TPUSetInterrupt(channel); → TT8 library function, TT8C_Man.pdf sec 5 page 40
TPU_InterruptEnable(channel); → TT8 library function, TT8C_Man.pdf sec 5 page 42

Processes:

Clear IR interrupt and disable it to reset

Reset "loop_counter" to **-1**. Necessary because the first action in MainTimingInt is to increment loop_counter - since loop_counter is an *index* it must first refer to the "0th loop."
Call MainTimingInt(); which begins the OS cycle at stage 1 (index 0)
Re-enable the IR interrupt
Notes: none

B.1.3.8 InitTACH

Inputs: none
Outputs: none
Global variables accessed: none
Other references:
TPU_TACH (digital I/O pin receiving signal/data from tachometer)
TPU function: FQM → "tpupn03 - FQM Function.pdf" in TT8 App Notes
Processes: Uses process as defined in Motorola spec sheets to detect and translate signal coming in over pin TPU_TACH. Defines recognition of signal, initializes and enables channel.
Notes: Refer to TT8 app notes for more information on function setup and PRAM offsets.

B.1.3.9 GetTACH

Inputs: none
Outputs: returns type *float* (tach reading)
Global variables accessed: old_tach (RW speed detected on previous cycle)
Local variables: *float* speed (speed of RW - init to 0.0); *int* dir (direction of RW spin)
Other references:
TPUGetPin(channel); → TT8 library fuction, TT8C_Man.pdf sec 5 page 41
TPU_TACH (digital I/O pin receiving signal/data from tachometer)
TPU_TACH_DIR (digital I/O pin receiving *direction* data from tachometer)
Processes:
Note direction of RW (read TPU_TACH_DIR)
Read TPU_TACH and convert to RPM or rad/second
Smooth data - avg with last tach reading (old_tach)
Return new calculated RW speed as type *float*.
Notes: none

B.1.3.10 GetMetroData

Inputs: *unsigned short int* metrology_data[]
Outputs/Return: (modify metrology_data[])
Global variables accessed: metrology_data[] ←duplicate array in global/local; unresolved!!
Local variables: raw_data[], *int* loop (init to 0), numofloop
Other references: GetT() function called to pull in bytes from pin; strlen() C library function
Processes:
Initialize raw data holding array, loop vars
Call incoming-data function, GetT. (fills raw data array with bytes)
Converts bytes (*char*-type elements) of raw data into words (*short int*), metrology_data[]
Does this through a loop that combines every two elements of raw_data into one of metrology_data; use "8-bit shift" left to hold places; then express bitwise #as decimal for *int*.

Notes: Sec. B.1.3.10 describes how this modules *should* proceed. In fact, at time of print, this version of the code has been modified to work without incoming metrology data – so instead of using this function to call GetT and work with a data stream off the MET_CHAN TPU pin, we set the elements of metrology_data[] to initial values and checked that these values came through avionics/comm. correctly during initial debugging of IT-2A.

B.1.3.11 GetT

Inputs: *char* data[] (uses this because data coming across pin is 1 byte at a time)
Outputs: (modified raw_data)
Global variables accessed: none
Other references: TSerByteAvail and TSerGetByte (TT8C_Man.pdf, section 5)
Processes:

Use the TT8 library functions to pull the bytes coming across the metrology-avionics TT8 connection (MET_CHAN = TPU pin number). Fills raw data array with as many bytes as are available. Flushes input at completion (detects no more bytes available).

Notes: This is a reusable function called currently only by GetMetroData

B.1.3.12 ClearString

Inputs: *char* data[]
Outputs: (modified data[])
Global variables accessed none:
Other references: C library function strlen(array[]) (return string length of array of type *char*)
Processes:

Empties the string data[] - sets all elements to NULL.

Notes: This is a reusable function. Currently *should* be called by GetMetroData to prevent excess old data from interfering with new measurements. In fact it is not used...

B.1.3.13 CreateLocalPVA

- get a timestamp (based on the stopwatch timer started just before entering this mail loop)

Reference: TT8C_Man.pdf section 5 page 38 (`StopWatchTime()`)

- read in local attitude/rate and feedback data from the gyro and tach (hardware local to avionics board, not connected to metrology TT8).

Reference:

- convert all variables to the same accepted format (unsigned short int, in preparation for comm transmission); this is difficult because the timestamp access function returns an unsigned long-type value, and the gyro-read function returns a float-type value.
- enter the timestamp, metrology data and local attitude/rate data into the PVA (`PVA_Local`).

B.1.3.14 SendH

Inputs: *char* cs[]
Outputs: (array *not* modified)
Global variables accessed: none
Other references: SerPutByte = get byte off serial line (in TT8_Man.c section 5)
Processes: Use TT8 library function SerPutByte to transmit a single byte over the *serial* channel, the avionics output to the DR2000 communication hardware (hence send"H").

Notes: There is some confusion over data transmission. If you try to send more than 1 byte at a time, it appears that anything past the first byte is lost (ie.not put on the hardware for transmission),

B.1.3.15 GetH

Inputs: *char* data[]
Outputs: (modified data[])
Global variables accessed: none
Other references: SerByteAvail, SerGetByte (see SendH descriptions & sec 5 of TT8C_Man.pdf)
Processes: Checks to see if a byte is available on the serial channel; then brings it in and stores in raw data array.
Notes: see SendH for complementary function.

B.1.3.16 Transmit_PVA

For information on this communication function, see section B.2 Communication Software.
NOTE: This *replaces* the baseline function SendH, which contained no comm. code.

B.1.3.17 Fetch_PVA

For information on this communication function, see section B.2 Communication Software.
NOTE: This *replaces* the baseline function GetH, which contained no comm. code.

B.1.3.18 CreateMSA <SJS & MAS>

Variables: float MSA[], unsigned short PVA_Local[], unsigned short PVA_Remote[].
Processes:

Converts the raw data from PVA_Local[] and PVA_Remote[] from the raw unsigned short data to float.

Check function: ensure that the PVA distances are accurate.

Previous MSA's velocities are taken
Multiply those by dt.
Whichever PVA distance is closer is chosen for the MSA

Notes: Table B-X lists the elements in *float* MSA(13):

Table B-8: Elements in the Master State Array (MSA)

MSA[i]	*Function*
`MSA[0]`	timestamp for MSA creation
`MSA[1]`	x position of vehicle 0
`MSA[2]`	x velocity of vehicle 0
`MSA[3]`	y position of vehicle 0
`MSA[4]`	y velocity of vehicle 0
`MSA[5]`	angle of zero vehicle axis with respect to 0 veh axis --> frame of reference
`MSA[6]`	vehicle 0 rate (gyro measurement)
`MSA[7]`	x position of vehicle 1
`MSA[8]`	x velocity of vehicle 1

`MSA[9]`	y position of vehicle 1
`MSA[10]`	y velocity of vehicle 1
`MSA[11]`	angle of 1veh axis with respect to 0 vehicle axis
`MSA[12]`	vehicle 1 rate (gyro measurement)

Local variables used (type *float*):

MSA_time,
MSA_x0,
MSA_y0,
MSA_x0dot,
MSA_y0dot,
MSA_ang01,
MSA_ang0dot,
MSA_x1,
MSA_y1,
MSA_x1dot,
MSA_y1dot,
MSA_ang10,
MSA_ang1dot,
MSA_ang00,
dt,
dx1,
dy1,
MSAdist0,
MSAdist1,
Diff_PVA_MSA0,
Diff_PVA_MSA1,
PVA_sec,
PVAmsec_raw,
PVAmsec,
PVA0_met_time,
PVA0_ang1,
PVA0_dist1,
rategyro0,
rategyro1,
PVA1_dist0,
PVA1_ang10.

Notes:
The digits 1 or 0 in (e.g.) PVA1_ang10 imply the angle ccw *from* one vehicle (0 = master or central (NOT necessarily local), 1 = removed (base coordinates are non-zero).

B.1.3.19 FloatToInt <SJS>

Inputs: none
Outputs/Return: none
Global variables accesses: none
Internal variables: *int* counter, flag, intvar; *float* var;
Processes:

Takes a float value, and converts it to a value that can be read as an int.

Determines if the value is positive or negative.

Creates the variable int intvar, which is simply the truncated value of the MSA[] reference value.

Includes if( ) statements for all the different possible orders of magnitude and breaks down the float value into mantissa and exponent.

The float value is converted into a string of sixteen bits where the first number is the sign of the value (1 for negative, 0 positive), the next twelve for the mantissa, and the last three for the exponent.

B.1.3.20 BitwiseFun <SJS>

Inputs: none
Outputs: none
Global variables accessed: none
Internal variables: *int* first_half, second_half, result;
Other references: none
Processes: Bitwise arithmetic is performed to convert the float value into a string of sixteen bits where the first number is the sign of the value (1 for negative, 0 positive), the next twelve for the mantissa, and the last three for the exponent.
Notes: none

B.1.3.21 ConvertMSA <SJS>

Inputs: *float* MSA[]; int result;
Outputs/Return: *float* MSA[];
Global variables accessed: none
Local variables: *int* IntMSA[], *int* i;
Other references: none
Processes: Takes each float value in MSA[] and converts it into a string of 16 bits by calling the functions FloatToInt() followed by BitwiseFun().
Notes: none

B.2 Communications Software (JEU)

B.2.1 Packet Structure

The DR2000 Protocol Packet definition sets the overarching construction of the communications system packet structure. The DR2000 Protocol, shown in Table 0-A, defines the packet header, the declaration and allocation of data bytes, and the use of built-in error checking and correction routines. The communications team created a virtual network layer for integration of the DR2000 technology directly into the EMFFORCE project. This layer constructs a system specific network using secondary headers and TDMA (Time Division Multiple Access)-like timeslots to enhance the packet definitions hardwired into the DR2000s.

Table B-A: DR2000 Protocol Packet Definition (courtesy of the RFM DR2000 Manual)

Primary Packet Header					Data	Error Checking	
To Address	From Address	Packet Number	Command	Length	Data	Frame Check 1	Frame Check 2
1 Byte	1 Byte	1 Byte	1 Byte	1 Byte	n Bytes	1 Byte	1 Byte
0-255	1-255	1-255	3-239	1-255	0-255	0-255	0-255

B.2.1.1 Header

The primary packet header contains information used by the DR2000 communications transceiver. As per DR2000 Protocol specifications, the packet header contains:

- To Address (one byte)
- From Address (one byte)
- Packet Number (one byte)
- Command (one byte)
- Length (one byte)

For reference, the node assignments for our system are as follows:

- 0x31 Node 1: GUI Ground Station
- 0x32 Node 2: Vehicle 0
- 0x33 Node 3: Vehicle 1
- 0x34 Node 4: Vehicle 2

To Address: The “To Address” tells the DR2000 transceiver which node in the network to transmit to. The hex value of ‘0x00’ instructs the receiver to broadcast the packet to all nodes in the network. To transmit to a single node, this value should be set to the hex value representing that node. For example, in our system the hex value ‘0x31’ represents node 1 (also ASCII character value ‘1’). This value is automatically stored in the flash memory of the DR2000.

From Address: The “From Address” tells the other nodes which node in the network sent the packet. Once again, the value must be in hex, with the same rules regarding the address values applying except for the broadcast value (a “From” broadcast is meaningless). The “From Address” value also is automatically stored in the flash memory of the DR2000.

Packet Number: The “Packet Number” header exists for two main reasons. First, it allows the application layer in the network to keep track of packets emanating from one source. Second, it allows the error-checking and correction software to request a repeat of a packet if necessary (the DR2000 supposedly makes use of an Automatic Repeat Request error-correction routine). Our system uses packet numbers with decimal values ranging from 48 (ASCII character ‘0’) to 57 (ASCII character ‘9’). These packet numbers recycle every ten packets.

Command: The “Command” value is currently set at 0x40. The actual function of the “Command” has not been determined. However, it could be used to transmit additional information about the packet (similar to a secondary header) as long as the software is adjusted to interpret the “Command” header. This may run into problems in the GUI since it uses a specific byte sequence to capture the desired information.

Length: The “Length” allocates the number of bytes used for the transmitted data. This value needs to be consistent with the flash packet size required by the DR2000.

NOTE: The flash packet size is the data length plus 5 (for the header).

B.2.1.2 Secondary headers

Secondary headers provide the functionality of a primary packet header (Please see the section on the packet header) but are contained within the body of the packet (data bytes). They are often used to provide information concerning the packet in addition to what is provided in the primary header. They are particularly useful for transmitting information about how the data should be interpreted or decoded.

The current system has a single secondary header that is used by the GUI to decide what information it should record. This header consists of a two-byte sequence of the ASCII character ‘6’ (54 decimal). When coded, the secondary header appears as:

```
TSerPutByte(OCHAN, 54);
TSerPutByte(OCHAN, 54);
```

When a packet enters the serial port of the ground station laptop, the GUI looks for a predetermined byte sequence. When that byte sequence is found, the GUI truncates the remainder of the packet and sends the truncated version off for analysis. The secondary header serves as that predetermined byte sequence.

It may be possible to use the "Command" value in the packet header as a secondary header. Be forewarned that significant problems will arise if the system becomes unstable and the header bytes are lost (which has happened before).

If the system complexity increases, it will be possible to implement APID (APplication IDentification) functionality by adding to the secondary header. If multiple packets need to be transmitted per vehicle per time slot, APIDs can be used to confirm the identity of the packets being transmitted. Furthermore, APIDs can enable implementation of a dynamic communications system

B.2.1.3 Data

The current communications system transmits and receives the following data structures:

- Vehicle 1 Primary Vehicle Array (PVA)
- Vehicle 2 PVA
- (optional) Vehicle 3 PVA
- Ground Station Operational Commands

Each Primary Vehicle Array captures the following information:

- The local vehicle metrology data timestamp in seconds
- The local vehicle metrology data timestamp remainder in milliseconds (the rest of the timestamp after the number of seconds is truncated)
- The distance to the first remote vehicle as measured from the local vehicle
- The angle from the line of sight between the local vehicle and the first remote vehicle to the local "east" direction
- A timestamp to mark the creation of the local vehicle PVA in seconds
- A timestamp to mark the creation of the local vehicle PVA with the remaining milliseconds (the rest of the timestamp after the number of seconds is truncated)
- The angular rate of the local vehicle, scaled and type-cast
- The amperage of the first electromagnetic coil of the local vehicle, scaled and typecast
- The amperage of the second electromagnetic coil of the local vehicle, scaled and typecast

The addition of a third vehicle into the system generates the following additions to each PVA:

- The distance to the second remote vehicle as measured from the local vehicle
- The angle from the line of sight between the local vehicle and the second remote vehicle to the local "east" direction

The Ground Station operational command data structure has not yet been determined.

B.2.1.4 Error-checking and correction routines

The DR2000 employs a hardwired error-checking and correction routine known as ARQ, or Automatic Repeat Request. If a packet arrives in error, ARQ should request from the sending node that the offending packet be resent. The communications team has so far been unable to determine whether this function is actually working. To utilize ARQ, the packet definition includes two error-checking bytes in Frame Check Sequence (FCS) or Cyclic Redundancy Check (CRC) format (shown in Table 0-A).

B.2.2 GUI (ESS)

The GUI is simply a Graphic User Interface coded in Labview for the express purpose of real-time monitoring, control, and data logging of the system. After initialization, the GUI runs in an infinite loop until an error occurs or the user manually deactivates the virtual instrument.

B.2.2.1 Initialization

The GUI performs all the necessary functions to initialize the laptop communications port to receive data from the DR2000. This process is transparent to the user as long as the default values are used for system constants.

B.2.2.2 String Capture

The GUI waits a user-determined length of time for bytes to accumulate on the serial port. It then reads them in as a character string. This string is the ASCII interpretation of the byte data stream output by the DR2000.

B.2.2.3 String Parsing

If the sample time is set properly and the DR2000 is successful in receiving good packets, each character string will contain at least one good packet. The GUI, using the “match pattern” virtual instrument, will examine the string until it finds the user-determined Header Tag. Once the header tag is located, the virtual instrument takes a substring of user-determined length from the original character string and passes it to the next function.

- If no header tag is found, an empty string is passed.
- If multiple header tags exist (usually from multiple packets) only the first will be passed on
- If the number of bytes to be passed exceeds the data available, the rest will be zeros.

Ideally, one complete data packet will be passed as a character string to the next function.

B.2.2.4 Type Conversion

The character string needs to be converted to an unsigned integer array, as this is how it was passed over the RF link from the vehicle. The high and low bytes are interspersed by a "for" loop and case function, creating a new array of unsigned integers.

B.2.2.5 Unit Conversion

The variables passed over the communications system RF channel each have a different "actual" range prior to being scaled to unsigned integers. In order to see real-time data that makes sense, the scaled unsigned integers need to be converted back to their original data type and rescaled. This requires a conversion matrix with two columns and n rows, where n is the length of the integer array: the first is a binary variable that determines whether or not the variable needs to be converted to a signed variable; the second column is the scaling factor that converts the recentered integer into an actual float value in the appropriate units.

As of May 2003, this functionality has not been implemented.

B.2.2.6 Data Logging

The integer array is appended to the end of a spreadsheet file for later analysis.

B.2.3 Communication Procedures

The communications software consists of two main procedures that operate within the Avionics operating system. The communications system's responsibilities include:

- Taking in data from the other software modules within the local vehicle's operating system
- Encoding the data for transmission
- Transmitting the data to the other network nodes
- Receiving the data from the other network nodes
- Decoding the received data, and
- Outputting the appropriate arrays to the other software modules running within the operating system of the remote vehicles.

These responsibilities are divided into two procedures: transmit and receive.

B.2.3.1 Transmit

The transmit function takes the local PVA array composed of 16-bit unsigned integers and packages it into an array of 8-bit high and low bytes for transmission. The high byte is simply the upper 8-bit chunk of the 16-bit unsigned integer shifted bitwise into byte format while the low byte is the lower 8-bit chunk of the same 16-bit unsigned integer.

For example, let’s suppose we have an unsigned integer of decimal value 20,000. In binary, this unsigned integer becomes:

0100111000100000

Thus, the low byte is:

00100000

And the high byte is:

01001110

Note that the high byte is the original 16-bit value bit-wise “anded” (‘&’ in C) with 65280 decimal, or:

1111111100000000

And shifted right (‘>’ in C) by 8 bits.

The function then transmits this packaged array through the serial line connecting the TT8 with the DR2000 transceiver.

NOTE: The nature of the serial connection means that data can only be sent as bytes, which explains the necessity of packaging the data for transmission.

NOTE: The first bytes to be sent through the serial line to the DR2000 for **each** packet **must** be the five header bytes. If these are not sent through the serial line, the DR2000 will not send the packet.

IMPORTANT: *There must be a delay between the transmission of each byte; otherwise there will be problems with the DR2000’s*. The issue seems to be related to the DR2000 internal buffer. Apparently, the TT8 can send bytes to the DR2000 internal buffer faster than the communications board can read them. This mismatch in throughput will cause packet collisions. To avoid this problem, we have chosen a delay of 1 millisecond, which seems to be sufficient to keep the communications system working reliably with a high data throughput. The use of shorter delays has not yet been tested.

B.2.3.2 Receive

The receive function (aka ‘Fetch’) accepts the data packets from the other network nodes by grabbing each byte as it appears in the serial line between the TT8 and the DR2000. As a byte comes into the TT8, it is assigned as an element value in a temporary storage array (PVA_Remote_Packaged, for example). When all of the data has been collected, the function converts this array back into a 16-bit unsigned integer array referred to as one of the PVA_Remote arrays or the GUI_Command array, depending on the source of

the data. The other software modules can then retrieve these arrays as needed during the rest of the operating cycle (see Avionics for a description of the operating system).

B.2.4 Network Structure

B.2.4.1 Node definitions

The communications network node assignments for our system are as follows:

- Node 1: GUI Ground Station
- Node 2: Vehicle 0
- Node 3: Vehicle 1
- Node 4: Vehicle 2

B.2.4.2 Communications cycle

The communications cycle is but one portion of the overall operating system cycle. Within the communications time slot, each node has a certain amount of time in which to transmit its data. The breakdown follows in Table 0-B:

Table B.2-B: Communications Cycle Broken Down Into Time Slots

Time slot 1	Time slot 2	Time slot 3	Time slot 4
Node 2 transmit All others receive	Node 3 transmit All others receive	Node 4 transmit All others receive	Node 1 transmit All others receive

The Ground Station (node 1) communication is considered to be the lowest priority; hence it transmits last. Any commands sent by the ground station will be implemented in the next operating system cycle.

Each node broadcasts its data to every other node. Each vehicle requires information from all the other vehicles in order to make the appropriate control calculations. The only way to achieve this information flow is for each vehicle to broadcast their PVA. The command packets can then be directed to one or all of the vehicles. Since the current operating system allocates a fixed amount of time for each node to transmit, it makes sense to simply go ahead and broadcast all of the command packets, regardless of which communications node is the intended recipient.

B.2.4.3 Bandwidth usage

The DR2000s transmit at 57,600 baud, which is 57.7 kbps equivalently. This represents our system data rate (since only one transceiver can transmit at any given time).

Given the fixed packet size chosen for the current version of the communications system, we know that we have:

- 5 header bytes (40 bits)
- 32 data bytes (256 bits)
- 2 CRC bytes (16 bits)
- 78 start/stop bits (one start bit and one stop bit for each byte sent)

This gives a total of 390 effective bits for one packet.

Our current communications network transmits at most 4 packets per cycle (one command packet, and at most 3 vehicle PVA packets). This means we transmit 1,560 bits for a cycle with maximum bandwidth usage. Given our system baud rate, we can calculate the cycle length in milliseconds:

$$\text{Cycle length} = \frac{bits}{cycle} * \frac{\sec onds}{bits} * \frac{1000ms}{\sec ond} \text{ (ms)}$$

= (1560 bits/cycle) * (1/57600 seconds/bits) * (1000 ms / second)
= 27.08 ms/cycle

Now we can calculate the number of communications cycles we can perform in a second:

$$\text{Cycles per second} = \frac{1000ms}{\sec ond} * \frac{1cycle}{27.083333ms} = 36.92 \text{ cycles/sec}$$

B.2.5 DR2000 Communication

B.2.5.1 DR2000 Commands

The DR2000 has several sets of very useful commands that should be kept handy at all times when dealing with the communications system. To use these commands, use the Microsoft Windows application HyperTerminal or another suitable serial port program such as Telix.

THESE COMMANDS ARE CASE-SENSITIVE.

The first set deals with *commands for the local DR2000*:

- To display the **current DR2000 configuration**

  $$s

  This command will cause the serial program to output configuration data that looks like:

RF mode OOK
To Address = 31
From Address = 32
Packet Size = 40
Range on
Rxout on

- To change the **"To Address"** (in hex!)

  $$TOADhh Valid digits: 0-f (00 – ff)

  For reference, the node assignments for our system are as follows:

  - 0x31 Node 1: GUI Ground Station
  - 0x32 Node 2: Vehicle 0
  - 0x33 Node 3: Vehicle 1
  - 0x34 Node 4: Vehicle 2

  REMEMBER: this value is stored in the flash memory.

- To change the **"From Address"** (in hex!)

  $$FRADhh Valid digits: 0-9, a-f (00 – ff)

  For reference, the node assignments for our system are as follows:

  - 0x31 Node 1: GUI Ground Station
  - 0x32 Node 2: Vehicle 0
  - 0x33 Node 3: Vehicle 1
  - 0x34 Node 4: Vehicle 2

  REMEMBER: this value is stored in the flash memory.

- To change the **"Packet Size"** (in hex!)

  $$SIZEhh Valid digits: 0-9, a-f (01 – ff)

  REMEMBER: this value is stored in the flash memory.

  The Packet Size value that is stored in the flash is set once. It is not something that can be changed dynamically due to how the DR2000s are constructed.

The second set deals with *commands for the remote DR2000s* (these commands will affect the remote DR2000 with the same address as the "To Address" used by the local DR2000):

- To change the **"To Address"** (in hex!)

  $&TOADhh Valid digits: 0-f (00 – ff)

  For reference, the node assignments for our system are as follows:

  - 0x31 Node 1: GUI Ground Station
  - 0x32 Node 2: Vehicle 0
  - 0x33 Node 3: Vehicle 1
  - 0x34 Node 4: Vehicle 2

  REMEMBER: this value is stored in the flash memory.

To change the **"From Address"** (in hex!)

  $&FRADhh Valid digits: 0-9, a-f (00 – ff)

  For reference, the node assignments for our system are as follows:

  - 0x31 Node 1: GUI Ground Station
  - 0x32 Node 2: Vehicle 0
  - 0x33 Node 3: Vehicle 1
  - 0x34 Node 4: Vehicle 2

  REMEMBER: this value is stored in the flash memory.

- To change the **"Packet Size"** (in hex!)

  $&SIZEhh Valid digits: 0-9, a-f (01 – ff)

  REMEMBER: this value is stored in the flash memory.

B.2.5.2 TT8 Serial Settings

The DR2000 connects to the TT8 via one the TT8 TPU Serial lines. The input channel (receive) is TPU channel 14 and the output channel (transmit) is TPU channel 13.

NOTE: The serial line baud rate MUST be set at 115.2 kbps. The modified DR2000's are designed to match an RS232 baud rate of 115.2 kbps.

B.3 Metrology Code Overview (OM)

The metrology code is written to handle various interrupts that are triggered by an internal clock timer and both the IR and ultrasonic receivers. The initialization of the code begins by defining global variables needed by the metrology system and initializing the interrupts. After these initial steps the metrology systems goes into an infinite loop and handles each interrupt.

B.3.1 Version documentation

The version of the code is kept tracked by the variable *CODE_VER*. This variable is initialized at the beginning of the code and needs to be updated each time the code is updated.

B.3.2 Include Files

The following table (Table B.3-A) provides a list of the included files. These files are needed for many of the TT8 functions to be used.

Table B.3-A: Include Files

File name	*Library Type*
`Stdio.h`	Standard C
`math.h`	Standard C
`String.h`	Standard C
`Tt8.h`	TattleTale Model 8
`Tt8lib.h`	TattleTale Model 8
`Tt8pic.h`	TattleTale Mode 8
`tpu332.h`	TattleTale Model 8
`dio332.h`	TattleTale Model 8
`Userio.h`	TattleTale Model 8

`Stdlib.h`	TattleTale Model 8
`time.h`	TattleTale Model 8

B.3.3 Input and Output Channels

The following channels (please refer to Table B.3-B) are defined for integration with the TT8 and the metrology hardware. The table includes the variable name within the code, along with a description and the pin number for the TT8.

Table B.3-B: Metrology I/O Channels

Identifier	*Name/description*	*TT8 Pin Type and #*
IR_RECV_CHAN	IR receiver in	TPU 0
US_RECV_CHAN1	US1 receiver in	TPU 1
US_RECV_CHAN2	US2 receiver in	TPU 2
US_RECV_CHAN3	US3 receiver in	TPU 3
IR_XMT	IR transmit	TPU 4
US_XMT	US transmit	TPU 5
US_TIMER_CHAN	Sequence timer trigger	TPU 6
US_TIMER_RCV	Sequence timer receive	TPU 7
SendCHAN	I/O to avionics TT8	TPU 8

B.3.4 Global Variables

The constants set here in Table B.3-C are used system wide. They identify what vehicle ID, which is required to determine when in the sequence to pulse IR and US signals. They also allow for time stamping IR and US signals for distance calculations and for time stamping the data sent to the avionics TT8.

Table B.3-C: Global Variables

Identifier	*Name/Description*	*Used in function*
vehicle_ID	Vehicle identifier	main(),
usSeq, usStateSeq	Sequence identifier	handleUSTimer(),
ArmLength	Length of metrology arm	xylocation(),
gIRArriveTime, gUSArriveTimeVec, usArriveTimes	Timestamp of IR and ultrasonic arrival times	handleIR(), handleUSTimer()
GcounterRate	Clock Rate	main(), handleUSTimer()
gIR_Usdelay	Delay time between IR and US transmit	handleUSTimer()
PeriodDelay	Delay between timing sequences	handleUSTimer()
centerDistance,centerAngle	Distance and angle to other vehicle	handleUSTimer(), xylocation()
Usdistance	Distance from US transmitter of second vehicle to US receiver	handleUSTimer()
TimeStampIR	IR receive timestamp	handleIR(),

B.3.5 Function Prototypes

Each task of the code is divided into a function. Each function is responsible for a specific task and is called upon when needed. A list of the function prototypes is included in Table B.3-D.

Table B.3-D: Function Prototypes

Function name	*Description*
void SetupIRRcv(void)	Setup IR receive interrupt
void SetupUSRcv(int CHAN)	Setup ultrasonic receive interrupt
void SetupIRXmt(void)	Transmit IR
void SetupUSXmt(void)	Transmit ultrasonic
void handleIR(void)	Handle the IR signal when interrupt is triggered
void handleUSTimer(void)	Handle the US Timer when the interrupt is triggered
void SetupUSTimer(void)	Setup the US Timer
void SetupUSTimerRcv(void)	Setup the US Timer receive interrupt
double xylocation(double distanceA, double distanceB, double distancC, int satNum)	Calculate the distance from center of vehicle to second vehicle

B.3.6 Metrology Design Overview

The metrology system currently uses data obtained from IR and ultrasonic sensors to calculate the relative distance and angle of each vehicle. Each vehicle is equipped with one omni-directional ultrasonic transmitter, three omni-directional ultrasonic receiver, four omni-directional IR transmitter arrays, and three IR receiver arrays. There are multiple IR transmitters in case one transmitter has a limited field of view due to other devices needed on the metrology system. Also, the power of the IR signal is limited to each IR transmitter, by adding an array of transmitter we are able to guarantee the signal can reach the receiver of the other vehicles. There are multiple IR receivers because the receivers do not have a 360° field of view. The three receiver arrays allow us to cover the full field of view required for the system to work properly. The system then uses the position of the ultrasonic receivers and the time difference between the IR and ultrasonic receivers to calculate the relative distance and angle.

The current design of the system uses a Tattletale 8 processor (TT8), which is capable of handing ultrasonic and infrared transmitters and receivers. All code is written in C, which is easily uploaded on the TT8 via a serial connection from a PC. The timing sequence shown in Figure B.3-A was created to map out the sequence of events for the metrology system.

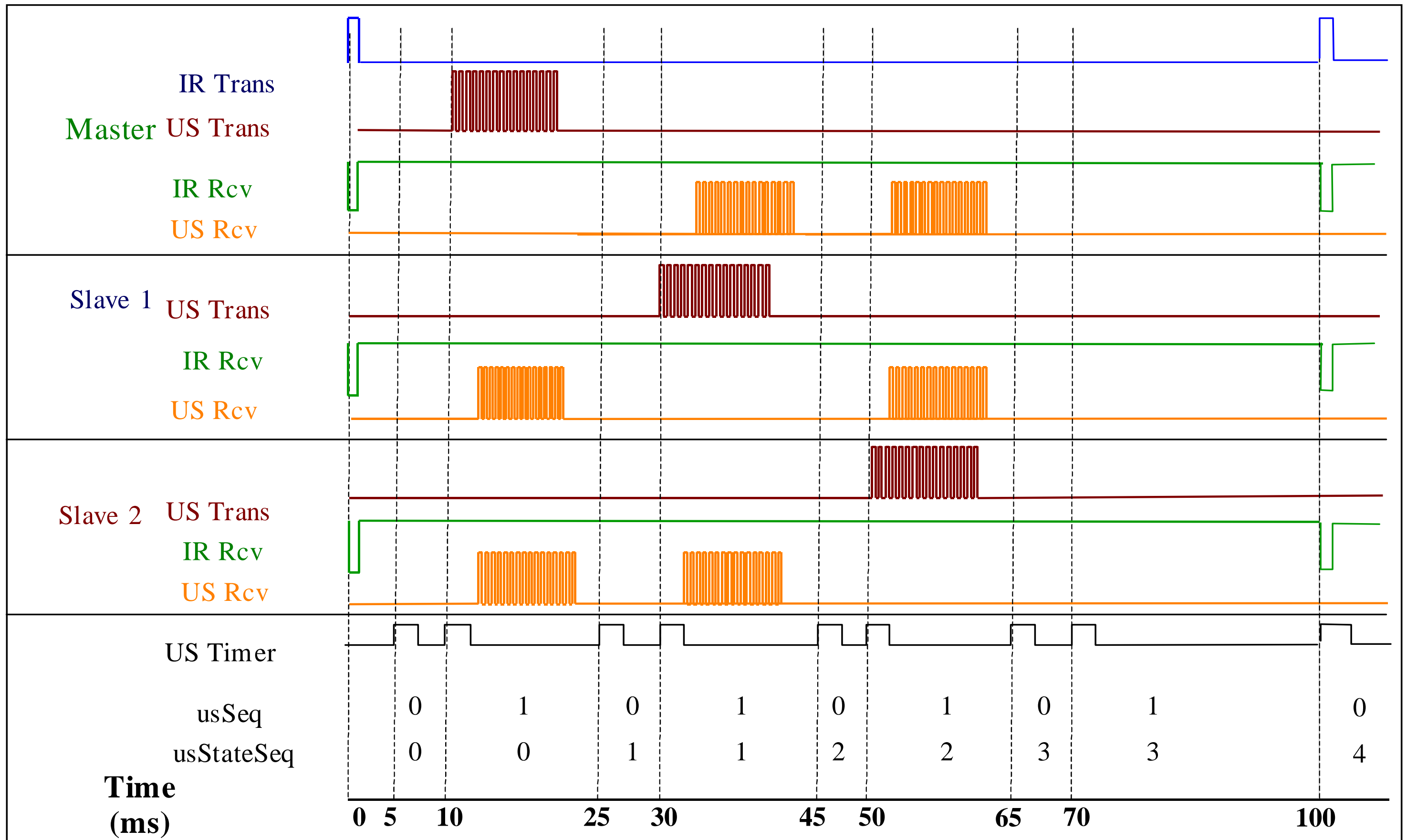


Figure B.3-A: Metrology System Timing

The first event begins with the master vehicle. Each vehicle is assigned master, slave 1, or slave 2 prior to the test. This label is available to each vehicle and can be set prior to each test over the communications port or prior to software load. The master vehicle emits an IR pulse. Each vehicle then receives the pulse (assumed to be instantaneous since the speed of light is much greater than the speed of sound) and causes an interrupt to be triggered on the TT8.

The IR triggered interrupt begins a timer, usTimer, which sets the TPU pin high or low at given times. When the TPU pin is set high, a second interrupt is triggered causing the TT8 to do a sequence of events based on timing and vehicle ID. Two counters are used to track the sequence of events. One counter, usSeq, oscillates between 0 and 1, switching back and forth each time the timer interrupt is triggered. A second counter, usStateSeq, increments by 1 each time usSeq resets to 0.

The following flow chart (Figure B.3-B) is followed once each vehicle receives the IR signal. The first command after the IR is received is to setup the timer, usTimer, to initiate the next interrupt in 5ms. It also initializes the counters, usSeq and usStateSeq, to zero. Following that, it enters a loop, allowing two separate events to occur depending on the value of usSeq.

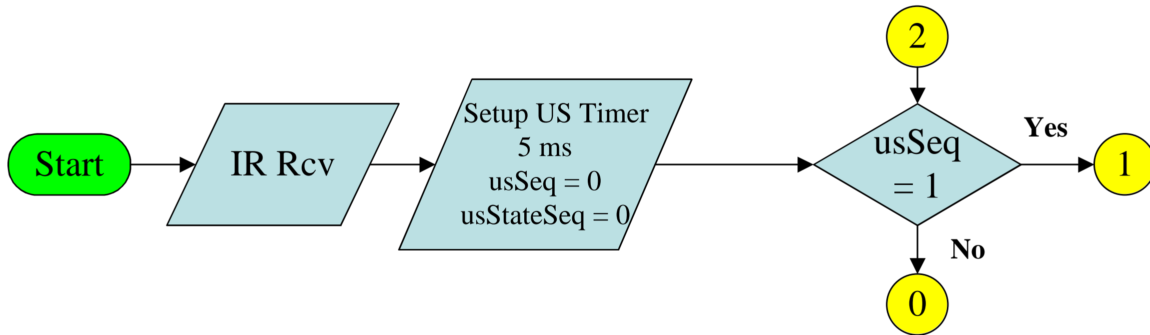


Figure B.3-B: Beginning Coding Sequence

When usSeq is zero, the following events occur (as seen in Figure B.3-C). The first check is to determine if the vehicle is in the beginning of the sequence (usStateSeq < 3) or the end. If usStateSeq is less than three, the vehicle then reads in the US receiver data. This data is the time stamp of the US signal arriving to the receiver, later used to determine the distance of the US signal. Following this, the code then checks to see if the state and vehicle ID match. If this is the case, the vehicle prepares to transmit an US pulse, otherwise it prepares to receive a US signal. Next, it resets the usTimer to initiates the next interrupt 15ms later. Finally, it increments usSeq by one and restarts the loop.

If the state is greater than three, a separate group of events occur. In this case usStateSeq and usSeq are reinitialized to zero. Next, a check occurs to see if the vehicle is the master. If this is the case, an IR pulse is emitted. The code then restarts the loop (including restarting the entire timing sequence).

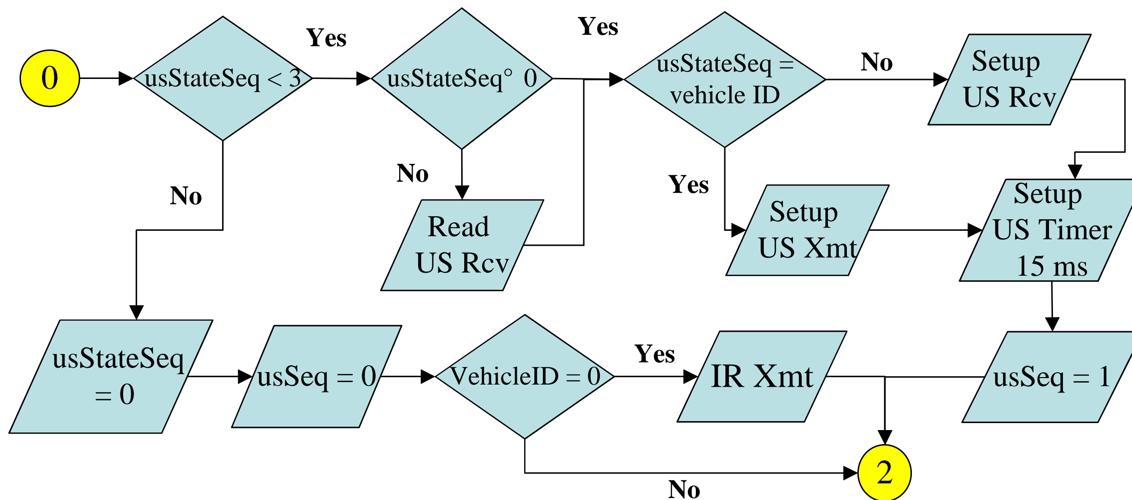


Figure B.3-C: Loop for usSeq = 0

When usSeq is set to one, the following events occur (as seen in Figure B.3-D). The highest priority task in the loop is the transmission of the US signal. This event is the first that takes place, with only a simple check to see if the vehicle is required to emit a US signal at that instant in the sequence. After that check, a second check is made to

identify the location within the timing sequence. If the sequence corresponds to a usStateSeq of less than three, the US timer is reset to 5 ms, usSeq is reinitialized to zero, and the code is looped to the beginning. If this not true, corresponding to the end of the US transmissions, the distance is calculated and sent to the main processor. In addition, the IR receiver is setup to receive another signal and the US timer is reset to 30 ms (to ensure the next IR signal is sent at 100ms). Finally, usSeq is reinitialized, and the loop is restarted.

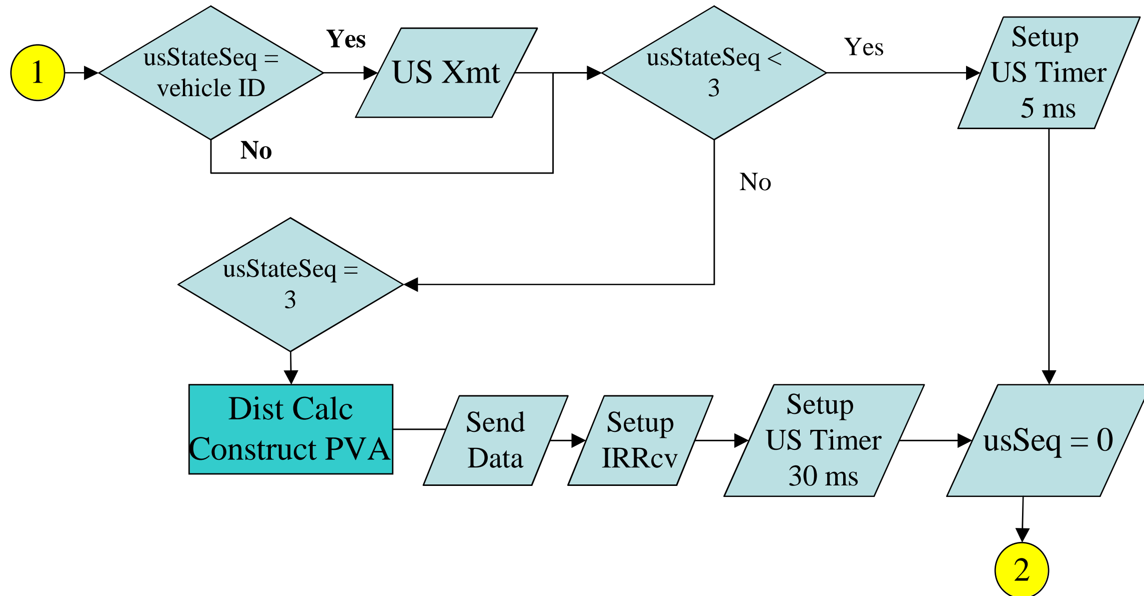
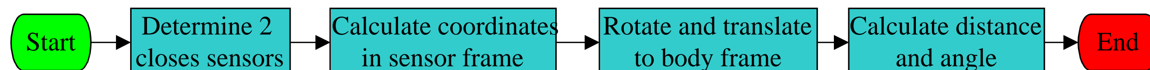


Figure B.3-D: Loop for usSeq = 1

B.3.7 Distance Determination Code

As a result of the transmitter and receiver code, each sensor will have a distance to each (3 sensors x 2 vehicles = 6 distances). These distances will then help the vehicle to determine a distance and angle from center of the vehicle sending the signal to the center of itself. The following algorithm is then used to determine the distance and angle. This algorithm is depicted graphically in Figure B.3-E.

Since only two sensors are needed to determine the two unknowns (distance, r, and angle, θ) the two sensors that read the closest distance are used. By eliminating the third sensor we are able to get an initial idea of where the signal came (reducing the signal origin to a certain range). A temporary frame of reference is set to the two sensors, with the origin at one sensor and the second sensor $(x_0,0)$ away. Since the distance is know to each sensor, you can determine the coordinates of the originating signal relative to the temporary frame. Once that coordinate is determined, the frame is then rotated and translated so that the frame of reference is centered at the center of the vehicle. The coordinates are the converted from Cartesian to polar so that an r and θ are known.



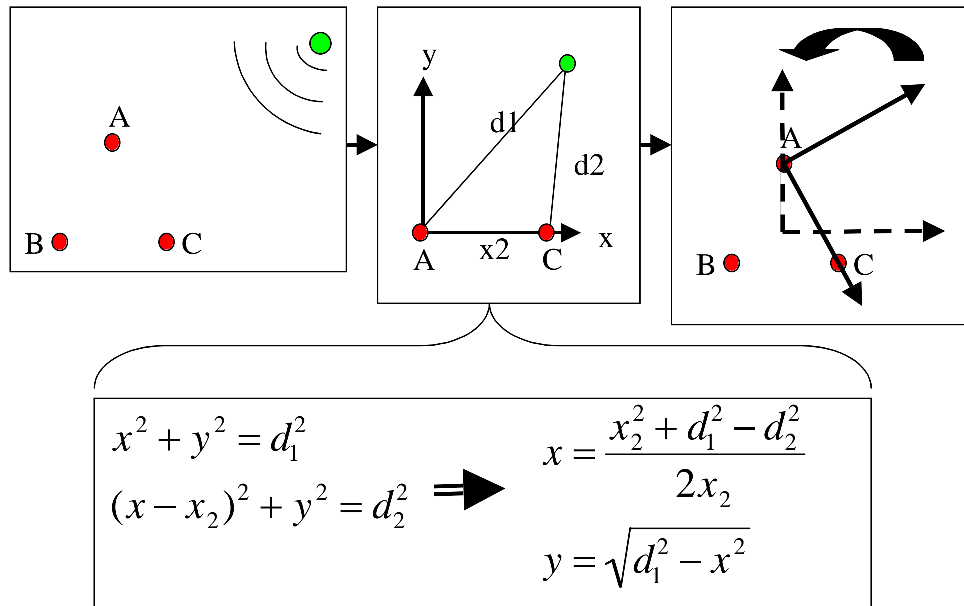


Figure B.3-E: Distance Determination Sequence

B.3.8 Metrology Software: Modules

Main()

The main function initializes the system to run. After printing out information (version number and the date and time compiled), the system runs into an infinite loop. To exit the loop, the avionics TT8 needs to send a start byte, which is a single byte containing the vehicle ID.

Next, the code then enables the interrupts (IR receive and the clock timer). Following this, it sends the first IR pulse (if the vehicle is the master vehicle). Then it enters an infinite loop and just handles all interrupts.

SetupIRXmt()

This function transmit the IR pulse. It uses the standard embedded function QOM. QOM is defined in the TT8 users guide.

SetupUSXmt()

This function prepares to transmit the ultrasonic pulse. The line of code that is commented out, enables the channel. This is needed to send the signal to pulse. The function uses the standard embedded function QOM. QOM is defined in the TT8 users guide.

SetupIRRcv()

This function prepares the IR receiver. The function uses the standard embedded function ITC. ITC is defined in the TT8 users guide. This allows the signal to trigger an interrupt (handleIR).

SetupUSRcv() This function prepares the ultrasonic receiver. The function uses the standard embedded function ITC. ITC is defined in the TT8 users guide.

handleIR() This function is used when the IR pulsed is received. The IR pulse triggers an interrupt that calls *handleIR()*. The function timestamps the IR pulse by calling a standard TT8 function *StopWatchTime()* (this allows us to timestamp the data for use later on in data analysis). It then initiates the timing sequence (US timer) that will initiate the events described in the software design. Finally it clears the IR interrupt, so that an unintended IR signal won't trigger the interrupt again (this is re-enabled in a later part of the code when the next pulse is expected).

SetupUSTimer() This function initiates an timer that will send a pulse at the end of the timer. This function uses an embedded function QOM. QOM is defined in the TT8 users guide.

SetupUSTimerRcv() This function prepares the IR receiver. The function uses the standard embedded function ITC. ITC is defined in the TT8 users guide. This allows the signal to trigger an interrupt (handleUSTimer).

handleUSTimer() This function handles most of the sequencing and calculating in the metrology system. The function uses the variables of *usSeq* and *usStateSeq* to keep track where the code is in the sequence of events. The section on software design describes the order of events.

xylocation() This function calculates the distance and angle to the center of vehicle using the data from the three ultrasonic receivers. The algorithm is described in the software design section.

B.3.9 Calibration Data

The data shown in Figure B.3-F was obtained to calibrate the hardware and correct for any error in the system. The first test was a distance calibration. In this test the angle was kept constant and the distance was varied. The graph shows the actual distance from the transmitter and receiver and the data received from the metrology TT8. The data was averaged and a standard error was calculated. Error bars are included to show the deviation between all the points that were averaged.

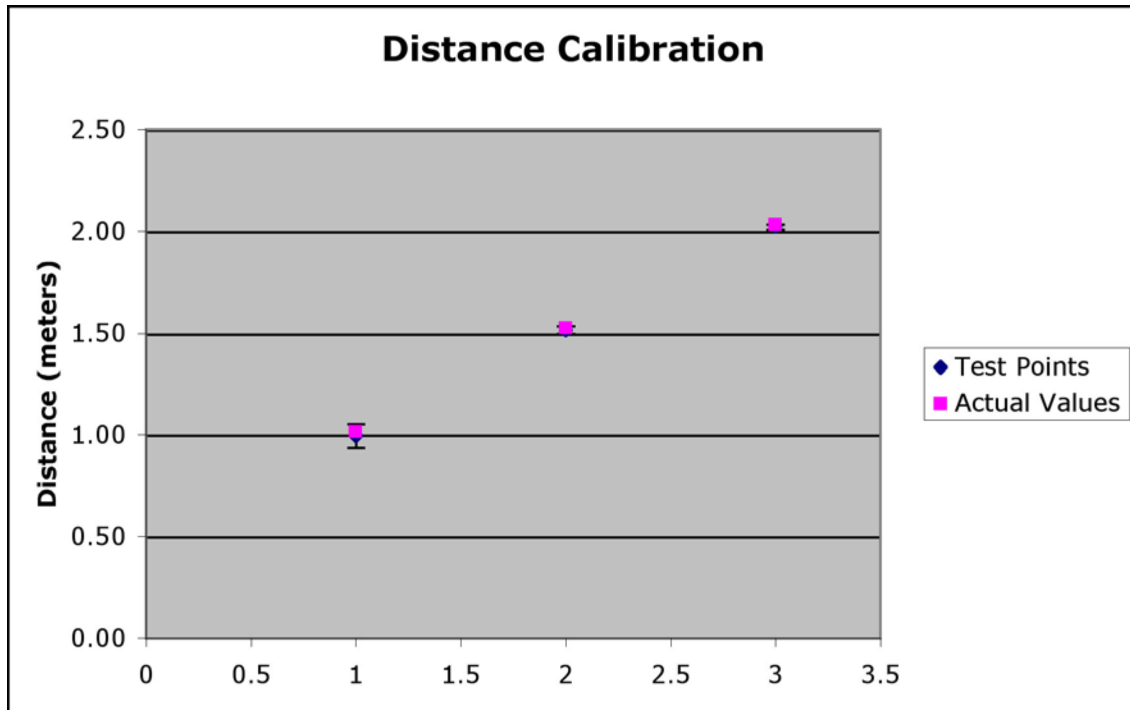


Figure B.3-F: Distance Calibration

The second test was an angle calibration. In this test, the distance is kept constant and the angle varied (at 30 degree increments). Figure B.3-G shows the actual angle from the transmitter and receiver and the data received from the metrology TT8. Again, the data was averaged and a standard error was calculated. Error bars are included to show the deviation between all the points that were averaged.

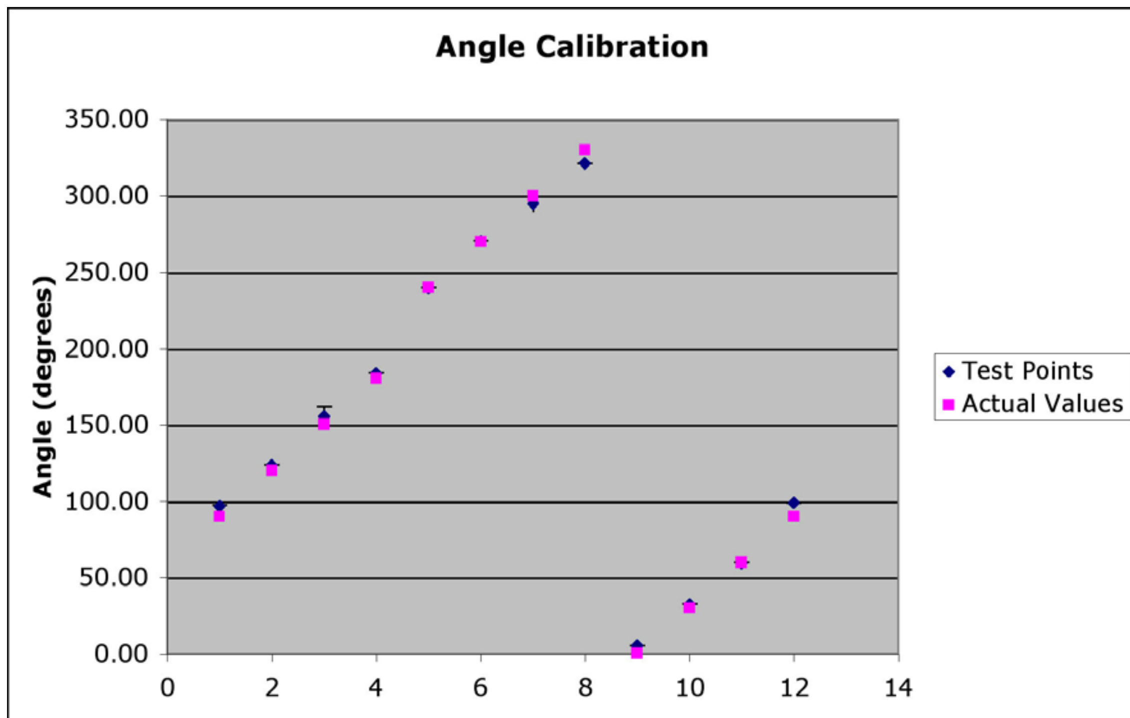


Figure B.3-G: Angle Calibration

The error between the actual values and the readings can be attributed to the value for the length of the arm of the metrology system. This value is included in the code and used in the angle calculation (it is also used for distance). A rough test to see how the length was critical to the sensitivity to the calculation was done. Although no data was collected, it was noted that moving the distance of one US receiver up to an inch varied the angle by up to 12 degrees. More test need to be done to get data on its sensitivity, but it should be noted that the distance from the center of the system to the ultrasonic sensor (defined as *ArmLength* in the code) needs to be accurate.