Application Note

AN2707
Rev. 0, 4/2004

Software Drivers for Tango3 RF Transmitter and Romeo2 RF Receiver ICs

By John Logan
8/16-Bit Division
East Kilbride, Scotland

Introduction

This application note describes a set of software drivers for the MC33493 RF transmitter (codename Tango3) and the MC33591/2/3/4 RF receiver ICs (codename Romeo2). The drivers are designed to allow a designer to quickly develop a new application using these RF ICs with minimum effort, or to add RF functionality to an existing design. The drivers are written in the C programming language. The Tango3 driver can be configured to use any HCS08 MCU. The Romeo2 driver can be configured to use any HC08 MCU with an SPI interface. Each driver allows the user to select different MCU I/O pins, timer channels, and clocking options, to allow easy implementation. Full source code listings and example applications are available from Motorola's web site at http://e-www.motorola.com.

The drivers provide the following features.

- Low CPU load and low MCU resource usage
- Transmission/reception of variable length messages with 0–127 data bytes
- Automatic checksum based error detection for each message
- Easy configuration options for carrier frequency, data rate, and other setup parameters.
- Support for networks with multiple transmitters/receivers

The drivers are primarily aimed at systems that use both Tango3 and Romeo2. However, each can be used separately, if required.

This document makes frequent references to the Tango3 and Romeo2 device datasheets; the reader should read these documents before using these drivers. Both datasheets are available for download from Motorola's web site at http://e-www.motorola.com.

Contents

Communication Concept

Tango3 and Romeo2 allow RF communications in the ISM (Industrial, Scientific, and Medical) bands 315 MHz, 434 MHz, 838 MHz, and 915 Mhz. Data rates up to 11 kbits/second are supported. This set of drivers provides a simple communications protocol to allow transfer of variable length messages with up to 127 bytes of data. The drivers support creation of networks with multiple receivers and transmitters.

Figure 1 shows a simple example lighting network with three lighting fixtures (each with a Romeo2 receiver) and one remote control (with a Tango3 transmitter). Each receiver is assigned a unique 8-bit identifier (ID). The transmitter can send messages to each receiver by changing the ID in the transmitted message. Additional transmitters and receivers could easily be added to this system.

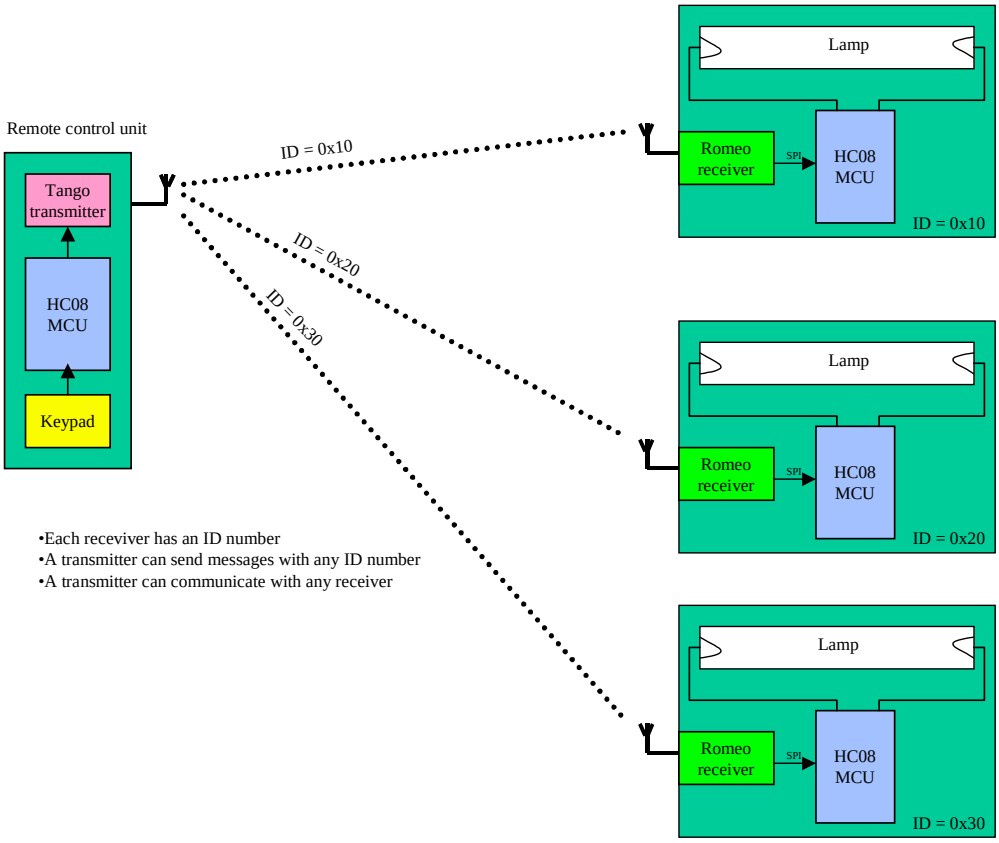


Figure 1. Simple RF Network

Message Formats

The drivers support the messaging formats defined in the Romeo2 datasheet. Communications using ID or tone signalling, with or without the header field are supported.

NOTE: *The Romeo2 driver uses Romeo2's on-board data manager hardware.*

The drivers extend the message formats shown in the datasheet, by defining length, data and checksum fields for each message.

A message contains the following fields.

Preamble — The Preamble is a fixed format field that allows Romeo2 to determine the timing of bits on the RF link. A Preamble field is required before each ID and Header field. See Romeo2 datasheet for more detailed information.

ID — Each Romeo2 device can be assigned an 8-bit ID number. It will only receive messages with this particular ID. This allows each Romeo2 device in an RF network to have a unique ID. A Tango3 transmitter can send messages with any ID. The ID field can also be used to implement Tone signalling, a simplified message format where each receiver uses the same fixed ID. See **Tone Signalling on page 7** for more detailed information.

NOTE: *The ID word must not contain the bit sequences '0110' or '1001'. These bit sequences are used as the header word field. See the Romeo2 datasheet for more detailed information.*

Header — The Header field is a 4-bit fixed format field. It notifies Romeo2 that message data is next. The header field is fixed to '0110' in this driver implementation. When Romeo2 receives the Header byte, it expects to receive the Length and Data fields next. It is possible to send messages with or without this field.

Length — The Length field is a byte containing the length of the Data field.

Data — The Data field comprises 0–127 data bytes.

Checksum — The Checksum field is a byte containing a checksum of the ID and data fields. The checksum value is calculated by adding all bytes in the ID and Data fields, using modulo 256 addition. (In MCU assembly language, this equates to adding the bytes using the 'Add with Carry' instruction.)

EOM (End of Message) — The EOM field is a fixed format field that indicates the end of a message.

NOTE: *Preamble, Header, Checksum and EOM are handled by the software drivers; the application programmer does not have to specify these fields.*

NOTE: *The Preamble and ID fields can be repeated.*

Sending Messages With Header Detect

Figure 2 shows a message frame transmitted by Tango3 and the received data that Romeo2 passes to the MCU using its SPI interface. Tango3 transmits a Preamble then the ID field. When Romeo2 receives a valid ID, it will wait to receive a Header field. When it receives a Header, it then expects to receive Length, Data, Checksum and EOM fields. While waiting for the Header field, it will ignore all other data. Note that its possible to have a delay between the ID field and the next preamble and header fields.

Using this message format, Romeo3 does not pass the ID field to the MCU on the SPI interface. It passes only the Length, Data and Checksum fields. This reduces the load on the SPI interface.

It is also possible to repeat the Preamble and ID fields multiple times. This is discussed below.

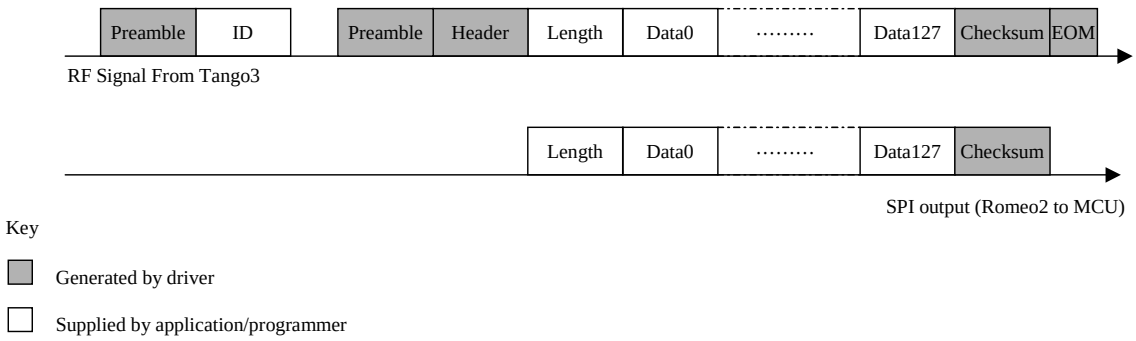


Figure 2. Message Format Using ID and Header

Sending Messages Without Header Detect

The header field is used to determine the start of the Length field. Romeo2 transmits all data, received after the header field, on its SPI interface, typically to an MCU. However, it is possible to send messages without using the header field.

Tango3 transmits a message containing Preamble, ID, Length, Data, Checksum and EOM fields. The ID field may be repeated to ensure that Romeo2 detects the ID, if it is using its Strobe oscillator as discussed below in **Reducing Power Consumption on page 6** (the number of repeats is programmable in the software driver). Once Romeo2 has detected the ID field,

it will pass all following data to the MCU via the SPI interface. See **Figure 3**. If the ID field is repeated, this will mean that ID bytes are also passed to the MCU.

This message format requires more CPU time to decode the received data, since it must handle the ID field.

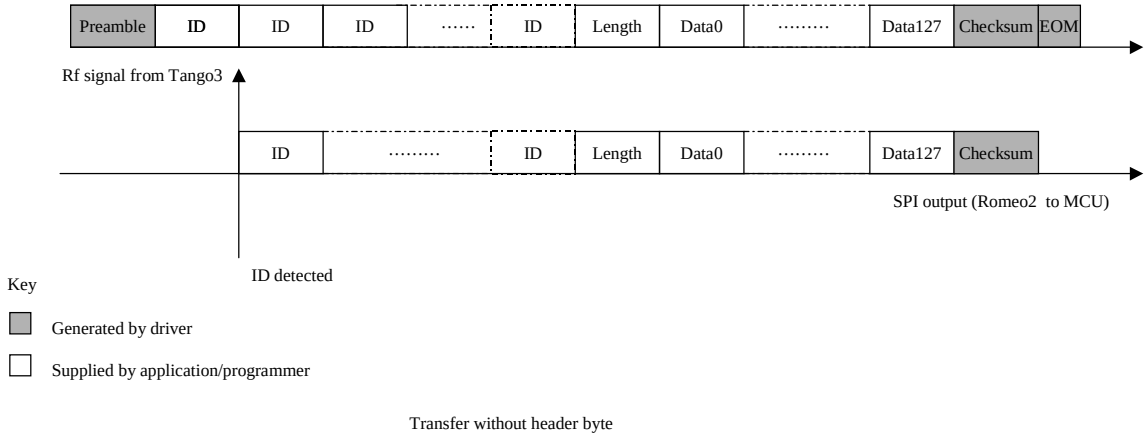


Figure 3. Message Format Without Header Detection

Reducing Power Consumption

To reduce power consumption in the system, the drivers can utilize two features of the Romeo2 device: the strobe oscillator; and the ability to repeat the ID field. The Romeo2 datasheet includes a full description of the strobe oscillator function.

The strobe oscillator function cycles Romeo2 between the very low power SLEEP mode and a RUN mode, thereby reducing the total current consumption. As mentioned previously, it is possible to repeat the Preamble and ID sections of the message. If Tango3 transmits a sequence of short Preamble + ID messages over a period longer than the strobe oscillator's SLEEP period, Romeo2 will detect at least one of these messages in RUN mode. When this has been detected, Romeo2 will override its Strobe oscillator, and will remain in RUN mode until it receives the remaining message fields. **Figure 4** shows the sequence.

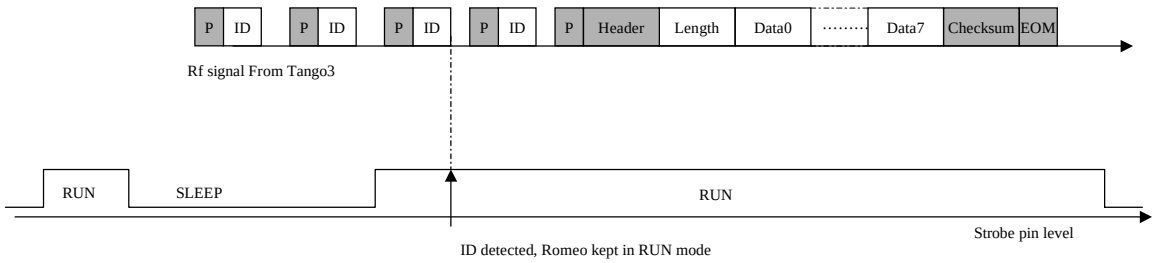


Figure 4. RUN/SLEEP Mode Cycling

Tone Signalling

Tone signalling is used in systems where all the receiver nodes must wake up and check each RF message. For example, in the simple lighting network shown in **Figure 1**, it could be desirable to control all the lights simultaneously. To do this, each node is assigned an ID of 0x00 or 0xff (i.e., the ID is all '1's or all '0"s). Then, all receivers will accept each message sent at the same time.

Figure 5 shows a message frame with Tone signalling. The Romeo2 IC supports Tones of any length greater than eight bits. This software driver allows the user to set the length of a Tone to multiples of eight bits by repeating the ID field multiple times

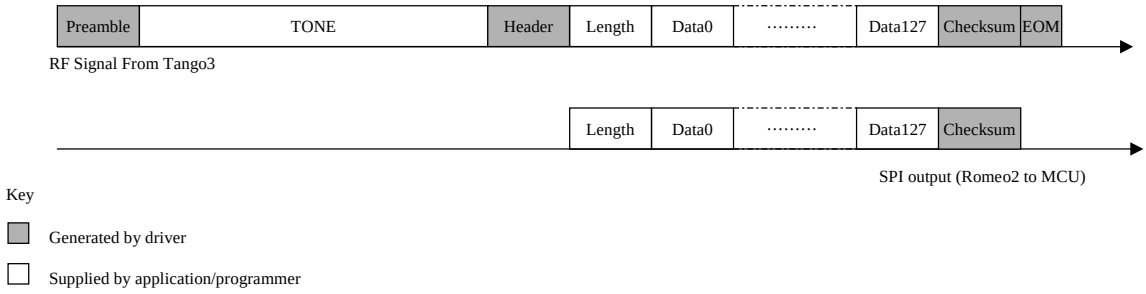


Figure 5. Message Format Using Tone Signalling

Message Encoding

Figure 6 shows the flow of data and the encoding stages for a message transfer.

For Tango3 to transmit a message, the application must provide the ID, length and Data fields of the message in a transmit buffer and call the correct driver routines. The software driver reads the message from this buffer and the message is encoded using the Manchester coding method prior to transmission. The message is then transmitted using Frequency Shift Keying

(FSK) modulation or On/Off Keying (OOK) modulation. Romeo2 receives the FSK/OOK signal, removes the Manchester encoding and passes the message to the software driver via the SPI interface. The software driver writes the message to a RAM buffer where the CPU can read the message. The Romeo2 driver must be correctly configured to match the message format, data rate and RF carrier frequency used by the Tango3 transmitter.

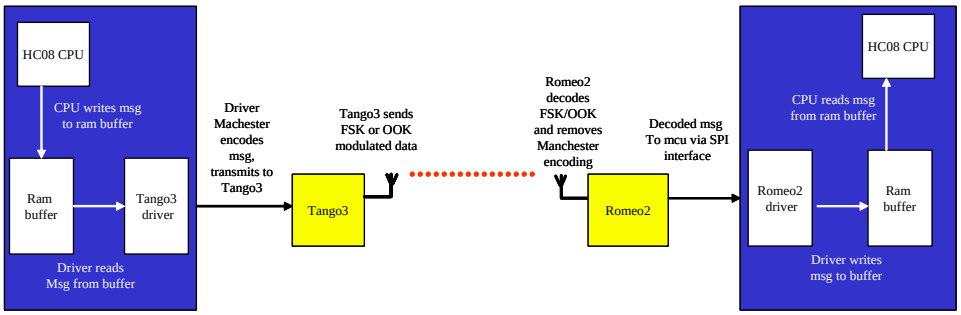


Figure 6. Data Flow and Message Encoding Steps in Message Transfer

Manchester Encoding

For Tango3 to transmit RF data, it must be supplied with a bitstream containing the data in Manchester encoded format. A Manchester encoded bit is represented by a sequence of two opposite logic levels. A '0' bit of data is encoded as sequence '01', a '1' bit of data is encoded as sequence '10'. **Figure 7** shows what will be seen on Tango3's DATA input when transmitting the data sequence '11001' using Manchester encoding. Note that there is always a level transition in the middle of a bit, but not always a transition on a bit boundary.

On the MCU, a timer I/O pin with an output compare function is used to generate each bit. The timer modulus (or timebase) is set to match the timebase of the Manchester encoded data. The output compare function is set to half the timebase. By controlling the level of the I/O pin when output compare occurs or the timer 'rolls over' to zero, the driver can generate the correct sequence. **Figure 7** shows the relationship between the timer counter value and the generated output.

Manchester encoding is performed by the Tango3 software driver.

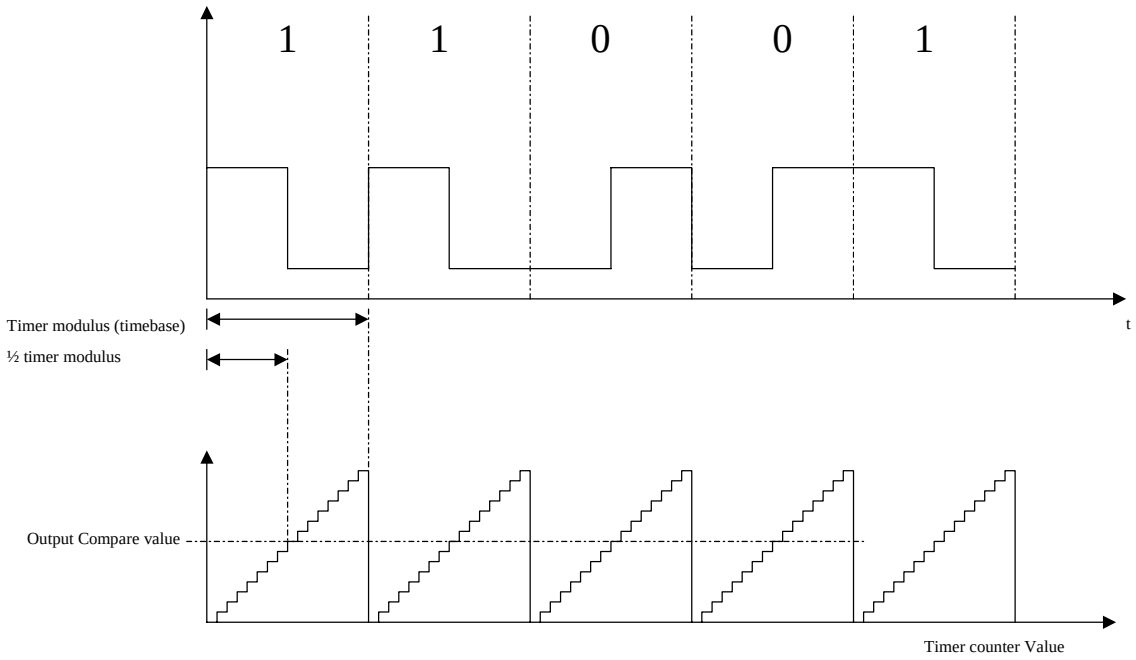


Figure 7. Bit Encoding Using MCU Timer Channel.

Bit Decoding

Romeo2's built-in data manager decodes the Manchester encoded data internally. It transmits decoded bits on its SPI interface.

Tango3 Driver

Tango3 Hardware Connections

Figure 8 shows the interface between the Tango3 IC and an MCU.

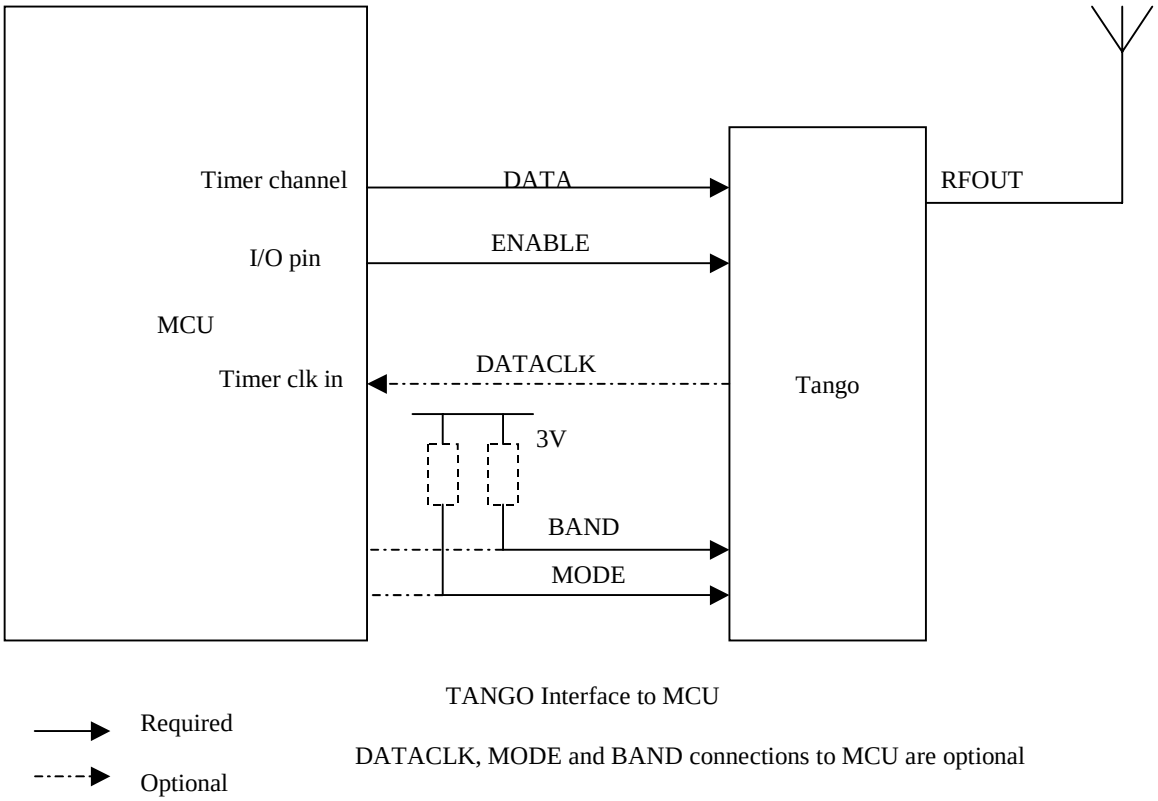


Figure 8. Tango3 Interface to MCU

A description of each connection between the MCU and Tango3 is given below.

DATA — The data to be transmitted over the RF link is passed to Tango3 on this line. It is encoded using Manchester encoding, as defined in the Tango3 datasheet, to be compatible with Romeo2's data manager. This data is generated using a timer channel on the MCU.

ENABLE — When this signal is at logic 1, the Tango3 IC is enabled and can transmit data. When this signal is logic 0, Tango3 is disabled and placed in a low power consumption mode.

DATACLK — This signal allows Tango3 to provide the MCU with an accurate clock signal, which can be used as an accurate timebase for generating data bits for transmission. This is useful with MCUs that use a low-accuracy clock source, such as, for example, an RC oscillator. When Tango3 is enabled (ENABLE = 1), DATACLK is active; when Tango3 is disabled (ENABLE = 0), DATACLK is at logic 0.

The software driver can be configured to use or ignore DATACLK.

BAND — This signal sets the operating band for Tango3, which defines the RF carrier frequency. This signal is usually hard-wired to a particular value but can also be controlled by the MCU. At logic 1, the RF carrier frequency is set to 32 times the Tango3 crystal frequency; at logic 0, the RF carrier frequency is set to 64 times the Tango3 crystal frequency.

The software driver can be configured to use or ignore BAND. BAND can be hard-wired to Vdd or ground if the carrier frequency is fixed.

NOTE: *Pull-up or pull-down resistors are required only if the MCU is required to override the hard-wired values.*

MODE — This signal sets the modulation mode for Tango3. This signal is usually hard-wired to a particular value, but can also be controlled by the MCU. When MODE is at logic 1, FSK modulation is selected; when MODE is at logic 0, OOK modulation is selected.

The software driver can be configured to use or ignore MODE. MODE can be hard-wired to Vdd or ground if the modulation mode is fixed.

NOTE: *Pull-up or pull-down resistors are required only if the MCU is required to override the hard-wired values.*

ENABLEPA — This signal is present on Motorola's Tango3 RF evaluation module. It allows the MCU to control an additional amplifier stage to boost the RF transmit power. When ENABLEPA is at logic 1, the power amplifier is enabled; when at logic 0, the power amplifier is disabled. The software driver can be configured to use or ignore ENABLEPA.

Tango3 MCU Resources

The Tango3 driver requires the following minimum MCU resources.

- One timer channel and its associated I/O pin used in output compare mode. The interrupt vector for this timer channel will also be used. The timer channel is connected to Tango3's DATA pin.
  The Tango3 driver will set the modulus value for the timer associated with the timer channel chosen.
- One I/O pin connected to Tango3's ENABLE pin. This allows the MCU to enable/disable Tango3, which can be useful for reducing current consumption in an application. Alternatively, the designer could tie the ENABLE pin directly to Vdd to permanently enable Tango3.

The following MCU resources may also be used.

- One timer channel configured as a clock input and connected to Tango3's DATACLK pin. (Some HC08 MCUs allow a timer pin to be used as a clock input source.) This allows Tango3 to provide the MCU with an accurate clock source, which can be used by the MCU to

generate accurate data on Tango3's DATA pin. This is especially useful with MCUs that use low-cost RC or internal clock sources, which are inherently inaccurate.

- One I/O pin connected to Tango3's MODE pin. This allows the MCU to select OOK or FSK operation. Alternatively, the designer could tie the MODE pin directly to Vdd or ground, to select OOK or FSK mode, respectively.
- One I/O pin connected to Tango3's BAND pin. This allows the MCU to select high or low band operation. Alternatively, the designer could tie the BAND pin to Vdd or ground, to select low or high band operation, respectively.
- One I/O pin connected to Tango3's ENABLEPA pin. (Note: This pin is provided for use with Motorola's Tango3 RF module, which features an on-board power amplifier.)

Tango3 Driver Description

This section provides a description of the Tango3 driver application interface and run-time services.

The Tango3 driver provides a set of runtime services using C function calls that allow the user to transmit messages. The services are listed below.

TangoInitialise — Configures the Tango3 driver (must be called when MCU resets).

TangoEnable — Enables driver (and Tango3 hardware) for transmission.

TangoDisable — Disables driver (and Tango3 hardware).

TangoDriverStatus — Returns current state of driver.

TangoSendPreamble_ID — Driver transmits message preamble and ID fields.

TangoSendData — Driver transmits Header, Length, Data and EOM fields.

TangoSendMessageNoHeader — Driver transmits a message frame with no header field.

TangoTimerInterrupt — Provides the driver with a link to the MCU's timer interrupt.

Messages are constructed in a RAM buffer prior to transmission. The driver can send messages with or without a Header field as describes in **Sending Messages With Header Detect on page 5** and **Sending Messages Without Header Detect on page 5**. **Figure 10** shows a flowchart for sending a message with a header field, **Figure 11** shows a flowchart for sending a message without a header field.

The Tango3 driver defines a transmission buffer in RAM. The MCU writes messages to the buffer, and the driver reads messages from this buffer and transmits them on the RF link. The buffer contains the message ID, Length and Data fields as shown in **Figure 9**. Note storage for the checksum field is not required. The Tango3 driver generates the checksum field internally and appends it to the message during transmission.

The size of the buffer can be programmed by the user, using the TANGO_MAX_DATA_SIZE parameter in the Tango.H header file. The buffer should be made large enough to receive the largest message being transferred. See **TANGO_MAX_DATA_SIZE on page 22** for details.

The application must not write to the transmission buffer while a message is being transmitted. This could lead to corruption of the transmitted message. The user can check if a transmission is in progress using the TangoDriverStatus() service. See **TangoDriverStatus on page 18**.

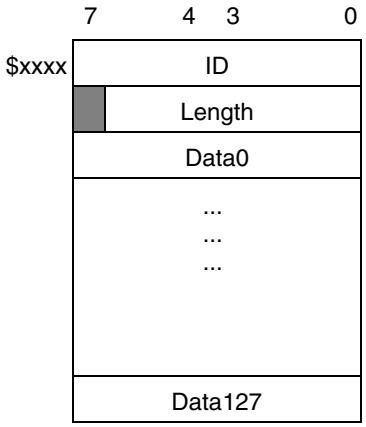


Figure 9. Tango3 Transmission Buffer

Internal processing of the driver occurs when the main application calls any of the run-time services, or after transmission of each bit when the driver is transmitting a message. Since transmission of each bit is controlled by a timer channel interrupt on the MCU, the user must link the TangoTimerInterrupt service to the timer channel interrupt. An example of this is shown in **Adding the Tango3 Driver to an Application on page 27**.

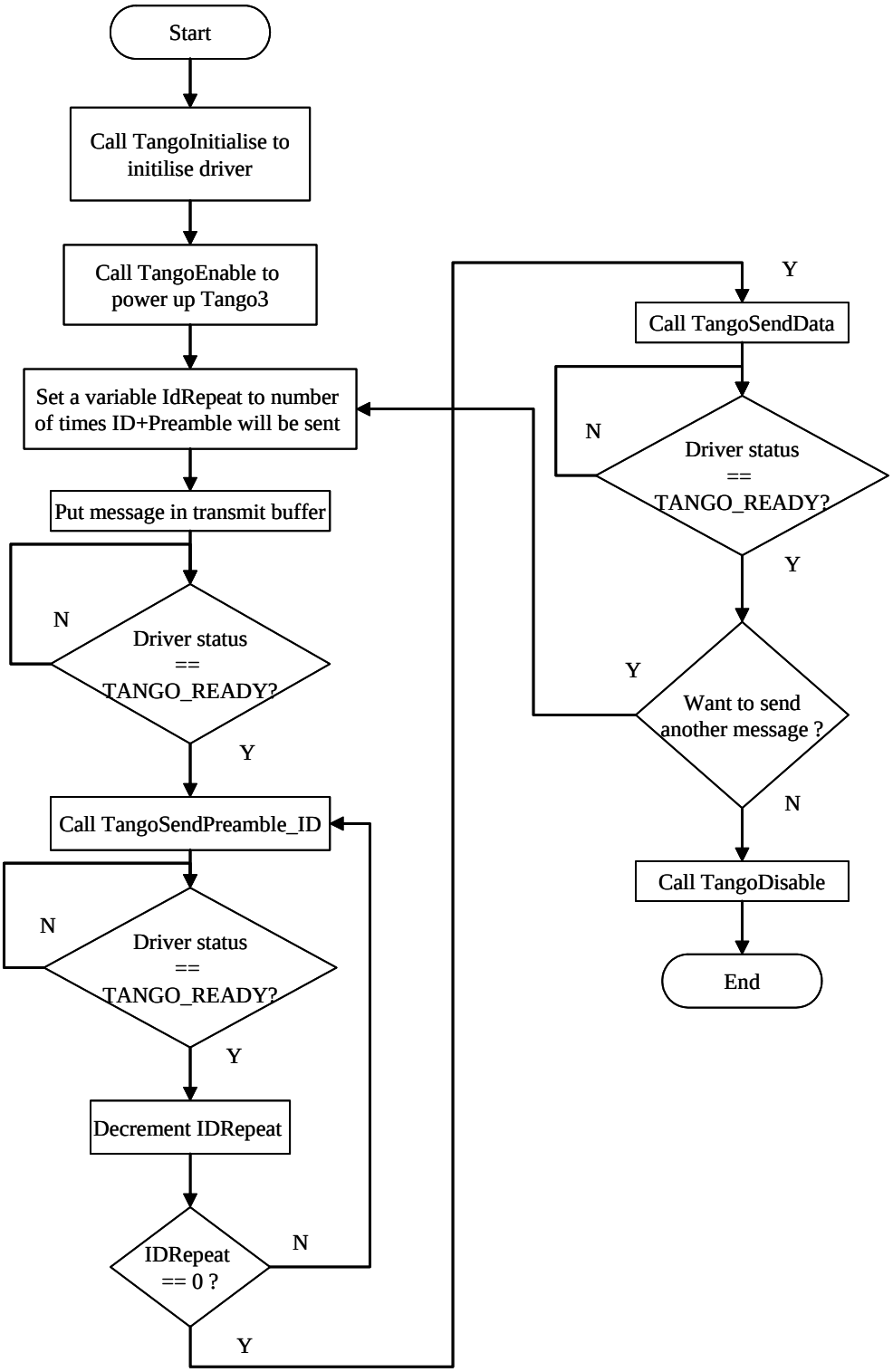


Figure 10. Sending a Message With a Header Field

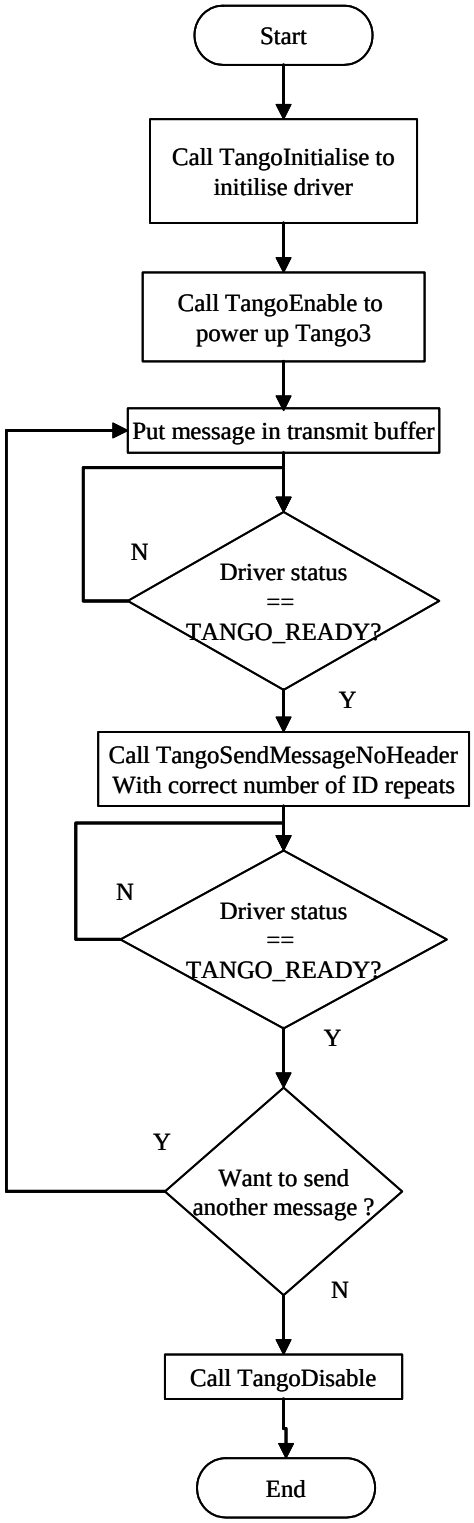


Figure 11. Sending a Message Without a Header Field

After the driver has been initialized, it can be in one of four states.

TANGO_DISABLED — Driver disabled, Tango3 IC is powered down

TANGO_READY — Driver enabled, Tango3 IC is powered up and ready to send data

TANGO_IN_ENABLE_DELAY — Driver enabled, Tango3 is currently powering up and is not available to send messages

TANGO_BUSY — Driver enabled, Tango3 is currently transmitting a message

Figure 12 shows the various states the driver will return when the TangoDriverStatus service is called.

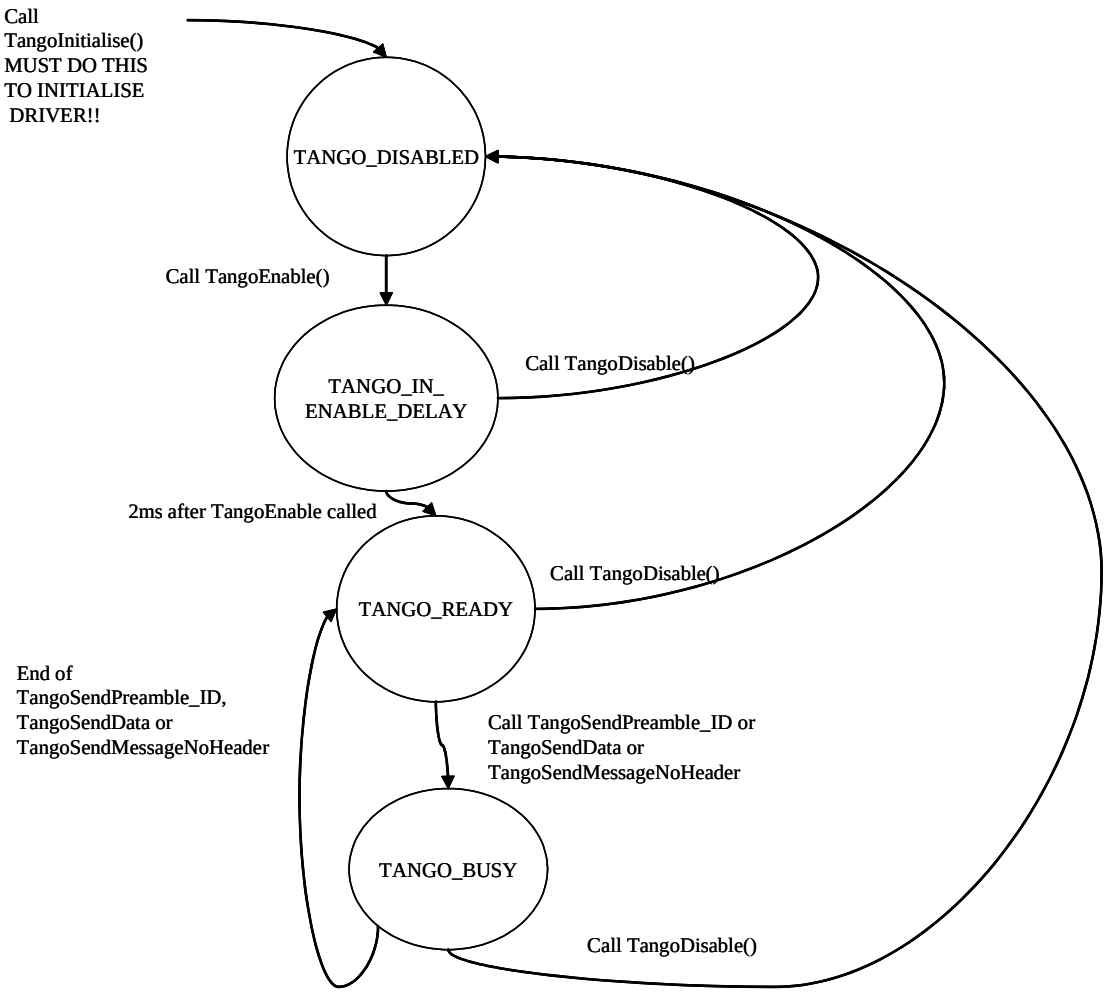


Figure 12. States Returned by the TangoDriverStatus Service

Tango3 Driver Services

This section provides descriptions of each service provided by the Tango3 driver.

TangoInitialise

Syntax: void TangoInitialise(void);

Parameters: None

Return: None

Description: The TangoInitialise service performs initialization of the Tango3 IC and software driver. It does not enable the Tango3 IC (i.e., the ENABLE pin is held low) to maintain low power consumption. It performs the following operations:

- Sets the driver status to TANGO_DISABLED
- Configures the MCU timer for use with Tango3 (Note that it does not switch the timer on)
- Configures MODE and BAND pins, if used

Notes: This service should be called before any other Tango3 driver services; otherwise, the result of any other Tango3 driver service and the Tango3 driver will be unpredictable.

TangoEnable

Syntax: void TangoEnable(void);

Parameters: None

Return: None

Description: The TangoEnable service powers up the Tango3 IC and starts a 2 ms time-out count. During the timeout, the driver status is set to TANGO_IN_ENABLE_DELAY. At the end of the 2 ms timeout, the driver status is set to TANGO_READY. At this point, Tango3 is powered up and ready to send data.

Notes: Typically, the application will call the TangoEnable service to start up the Tango3 IC. During the 2 ms timeout it can load a message into the transmit buffer and call the TangoStatus service to check if the 2 ms timeout has finished. When TangoStatus returns the value TANGO_READY, the application is ready to transmit the message.

TangoDisable

Syntax:	void TangoDisable(void);
Parameters:	None
Return:	None
Description:	The TangoDisable service sets the driver status to TANGO_DISABLED and powers down the Tango3 IC. If the TANGO_TIMER_DISABLE option is chosen in the Tango.h header file, the MCU timer will be switched off.
Notes:	If TangoDisable is called while a message is being transmitted, transmission will halt immediately.

TangoDriverStatus

Syntax:	unsigned char TangoDriverStatus(void);
Parameters:	None
Return:	• TANGO_DISABLED (Tango3 IC is powered down) • TANGO_READY (Tango3 IC is powered up and ready to send data) • TANGO_IN_ENABLE_DELAY (Tango3 is currently powering up and is not available to send messages) • TANGO_BUSY (Tango3 is currently transmitting a message)
Description:	The TangoDriverStatus service provides the application with the current status of the Tango3 driver.
Notes:	The application must not write to the transmit buffer when status is TANGO_BUSY. Doing so will result in incorrect data being transmitted.

TangoSendPreamble_ID

Syntax:	void TangoSendPreamble_ID(void);
Parameters:	None
Returns:	None
Description	The TangoSendPreamble_ID service triggers transmission of a message containing a Preamble field and an ID field. The ID is read from the Tango3 transmission buffer. The driver status is set to TANGO_BUSY during transmission of this message.
Notes:	This service and the TangoSendData service are used to send messages, using the format described in the section on **Sending Messages With Header Detect on page 5**. The service should be called only when the Romeo2 RX IC is configured to detect Header bytes in a message sequence (i.e., the option ROMEO_HE_VALUE is set to 1 in the Romeo.h header file).

TangoSendData

Syntax:	void TangoSendData(void);
Parameters:	None
Returns:	None
Description:	The TangoSendData service triggers transmission of a message containing Preamble, Header, Length, Data, Checksum and EOM fields. Length and Data are read from the transmit buffer. The checksum is calculated prior to transmission.
Notes:	This service and the TangoSendPreamble_ID service are used to send messages using the format described in the section on **Sending Messages With Header Detect on page 5**. The service should be called only when the Romeo2 RX IC is configured to detect Header bytes in a message sequence (i.e., the option ROMEO_HE_VALUE is set to 1 in the Romeo.h header file).

TangoSendMessageNoHeader

Syntax:	void TangoSendMessageNoHeader(unsigned char idRepeat)
Parameters:	idRepeat, a specified number of times
Returns:	None.
Description:	The TangoSendMessageNoheader service triggers transmission of a message containing Preamble, ID, Length, Data, Checksum and EOM fields. The ID field is transmitted idRepeat+1 times.
Notes:	This service is used to send messages using the 'No Header Detect' format described in **Sending Messages Without Header Detect on page 5**. The service should be called only when the Romeo2 RX IC is configured to not use header bytes (i.e., the option ROMEO_HE_VALUE is set to 0 in the Romeo.h header file).

TangoTimerInterrupt

Syntax:	void TangoTimerInterrupt(void)
Parameters:	None
Description:	This function controls the actual processing of the Tango3 driver. It is called by the interrupt vector of the timer channel used to generate data for the Tango3 IC. In the CodeWarrior parameter file, this interrupt vector must be directed to this function. This function MUST be included used to ensure proper operation of the software driver.

Tango3 Driver Configuration

The Tango3 driver has a static configuration at compile time. Its configuration cannot be changed during run time. The driver configuration is defined in a header file Tango.H. Configuration options are available for the following.

- Message format
- Message data rate
- Message modulation format (OOK or FSK)
- Carrier frequency
- MCU resources

These configuration options are set using a number of #define statements in the Tango.h header file. Using these #defines, the driver can be configured to run on any HCS08 MCU.

When starting a project using the Tango3 driver, the files ‘Tango.H’ and ‘Tango.C’ should be placed in the project directory, and a #include ‘Tango.H’ statement should be inserted in the main application file.

The Tango.H file contains 17 #define statements that must be configured to ensure correct operation of the driver. Thirteen #defines are mandatory, and four are optional, depending on the application’s hardware configuration. These are described below.

TANGO_TIMER_ADDRESS

Description: This defines the address of the timer status and control register, in the MCU's memory map. The timers on all HCS08 MCUs have the same layout of control registers. The driver uses this base address to access the timer control registers.

Values: Address in range 0x0000 - 0xffff

Example:

```
#define TANGO_TIMER_ADDRESS 0x30 /* 1st timer register at address 0x30*/
```

TANGO_TIMER_CHANNEL

Description: This defines the timer channel used to output data on the DATA line.

Values: Channel number in the range 0–15.

Example:

```
#define TANGO_TIMER_CHANNEL  1       /* Use timer channel 1 */
```

TANGO_MAX_DATA_SIZE

Description: This defines the maximum number of data bytes that can be transferred. This value is used to calculate the size of the message transmit buffer. (The transmit buffer will be TANGO_MAX_DATA_SIZE + 2 bytes.)

Values: Number in range 0–127.

Example:

```
#define TANGO_MAX_DATA_SIZE  8  /* Max size of data field = 8 bytes*/
```

TANGO_TIMER_CLOCK_SOURCE

Description: This defines the clock used to control the timer.

Values: 1 = Bus clock
2 = XCLK
3 = External clock source

Example:

```
#define TANGO_TIMER_CLOCK_SOURCE  3  /* Ext clk src for timer selected*/
```

TANGO_TIMER_CLOCK_SPEED

Description: This defines the clock speed in Hz of the timer if an internal clock is chosen. When an external clock is used, this definition can be deleted.

Values: Integer from 0 to MCU bus speed/4.

Example:

```
#define TANGO_TIMER_CLOCK_SPEED  2000000    /*  Timer clk = 2 MHz    */
```

TANGO_TIMER_PRESCALE

Description: This defines the prescaler value of the timer used to send data to Tango3. Generally this will be set to '1'. (Note: If an external clock source is being used, this value will be forced to '1' by the driver.)

Values: See datasheet for S08 MCU.

Example:

```
#define TANGO_TIMER_PRESCALE    1     /* Specify timer prescaler value       */
```

TANGO_TIMER_DISABLE

Description: This allows the driver to switch off the MCU timer when it is not required to drive Tango3. This can reduce power consumption. However, in some applications it may be required that the timer continue running. (Note: The driver will always start the timer when it is required.)

Values: 0 = Timer remains running after transmission of data.
1 = Timer is disabled after transmission of data.

Example:

```
#define TANGO_TIMER_DISABLE 1 /* Allows driver to turn off timer after use  */
```

TANGO_MODE_VALUE

Description: This defines the type of modulation used in RF transmissions: On Off Keying (TANGO_OOK) or Frequency Shift Keying (TANGO_FSK).

Values: TANGO_OOK denotes OOK modulation.
TANGO_FSK denotes FSK modulation.

Example:

```
#define  TANGO_MODE_VALUE   TANGO_OOK     /* OOK modulation*/
```

TANGO_BAND_VALUE

Description: This defines if Tango3 is used in high band or low band configuration.
High band: modulation frequency = crystal frequency/32
Low band: modulation frequency of crystal frequency/64

Values: TANGO_HIGH_BAND denotes high band.
TANGO_LOW_BAND denotes low band.

Example:

```
#define TANGO_BAND_VALUE   TANGO_HIGH_BAND     /* High band selected */
```

TANGO_CRYSTAL_FREQUENCY

Description: This defines the speed (in Hz) of the crystal used by the Tango3 IC.Typical values at supported RF frequencies are:

/* 315 MHz — 98400000 */
/* 434 MHz — 13560000 */
/* 868 MHz — 13560000 */

Values: Integer in range 0–10000000

Example:

```
#define TANGO_CRYSTAL_FREQUENCY 13560000    /* Crystal freq=13.56 MHz */
```

TANGO_DATA_RATE

Description: This defines the data rate in bps (before Manchester encoding).

Values: Integer in range 0–11000

Example:

```
#define TANGO_DATA_RATE      1000   /* Data rate = 1  kbps */
```

TANGO_ENABLE

Description: This defines the I/O pin used to control Tango3's ENABLE pin. If ENABLE is not controlled by the MCU in your system, delete this #define from the header file.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define TANGO_ENABLE        PTAD_PTAD0      /*Port A pin 0       */
```

TANGO_ENABLE_DDR

Description: This defines the data direction bit for the I/O pin used to control Tango3's Enable pin. If Enable is not controlled by the MCU in your system, delete this #define from the header file.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define TANGO_ENABLE_DDR    PTADD_PTADD0    /* DDR for Port A pin 0     */
```

The following #define statements are dependent on the hardware configuration of your system and may not be required. If not required, delete these entries from the header file.

TANGO_MODE

Description: This defines the I/O pin used to control Tango3's MODE pin. If MODE is not controlled by the MCU in your system, delete this #define from the header file.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define TANGO_MODE        PTAD_PTAD1      /* Port A pin 1  */
```

TANGO_MODE_DDR

Description: This defines the data direction bit for the I/O pin used to control Tango3's MODE pin. If MODE is not controlled by the MCU in your system, delete this #define from the header file.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define TANGO_MODE_DDR      PTADD_PTADD1    /* DDR for PortA pin 1    */
```

TANGO_BAND

Description: This defines the I/O pin used to control Tango3's BAND pin. If BAND is not controlled by the MCU in your system, delete this #define from the header file.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define TANGO_BAND          PTAD_PTAD2      /* Port A pin 2  */
```

TANGO_BAND_DDR

Description: This defines the data direction bit for the I/O pin used to control Tango3's BAND pin. If BAND is not controlled by the MCU in your system, delete this #define from the header file.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define TANGO_BAND_DDR      PTADD_PTADD2    /* DDR for PortA pin 2    */
```

Adding the Tango3 Driver to an Application

To add the Tango3 driver to an application:

1. Add Tango.h and Tango.c files to project (in CodeWarrior, right click on sources folder, then add files).
2. Add line #include ‘Tango.h’ to main application program file.
3. Add line 'extern unsigned char tangoTransmitBuffer();' to main application program file.
4. Decide which I/O pins in your application will control Tango3 functions. Modify the Tango.H file to link these pins to Tango3.
5. Decide which timer channel will be used to generate data for Tango3. Modify the Tango.H file to link these pins to Tango3.
6. Modify project parameter file to link timer channel to TangoTimerInterrupt service
7. Modify Tango.H file to define timer speed and other parameters.

The files are now added to the project.

Figure 13 is a screen shot of an application template with Tango3 files included and showing the main application program file with correct entries added.

Figure 14 is a screenshot of the project parameter file showing how to include the TangoTimerInterrupt function call. In this example, it is linked to timer 1, channel 0.

Figure 15 shows an example Tango.H header file. This has been configured for use with an MC9S08GB60 MCU. Tango3 is configured for 434 MHz operation with a data rate of 1 kbps. Timer 0, channel 1 is used on the MCU.

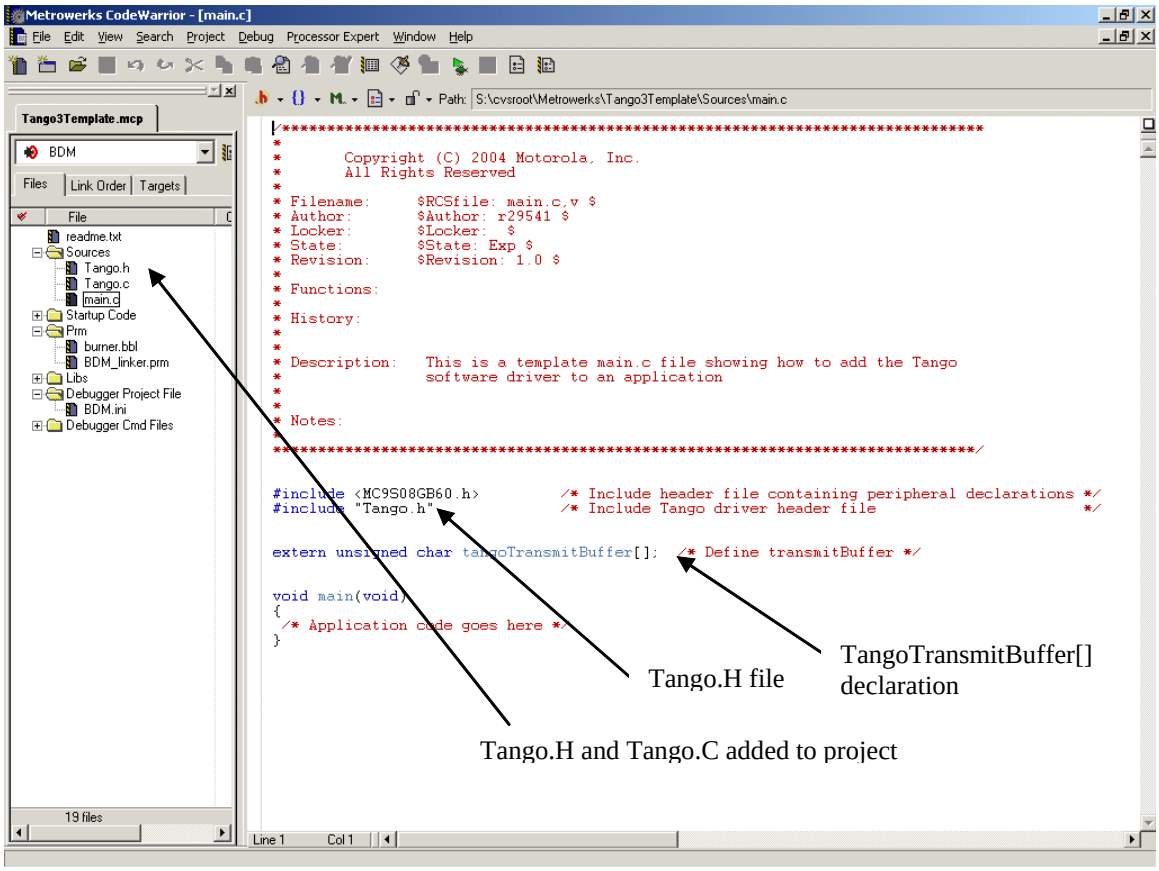


Figure 13. Application Template with Tango3 Files Included

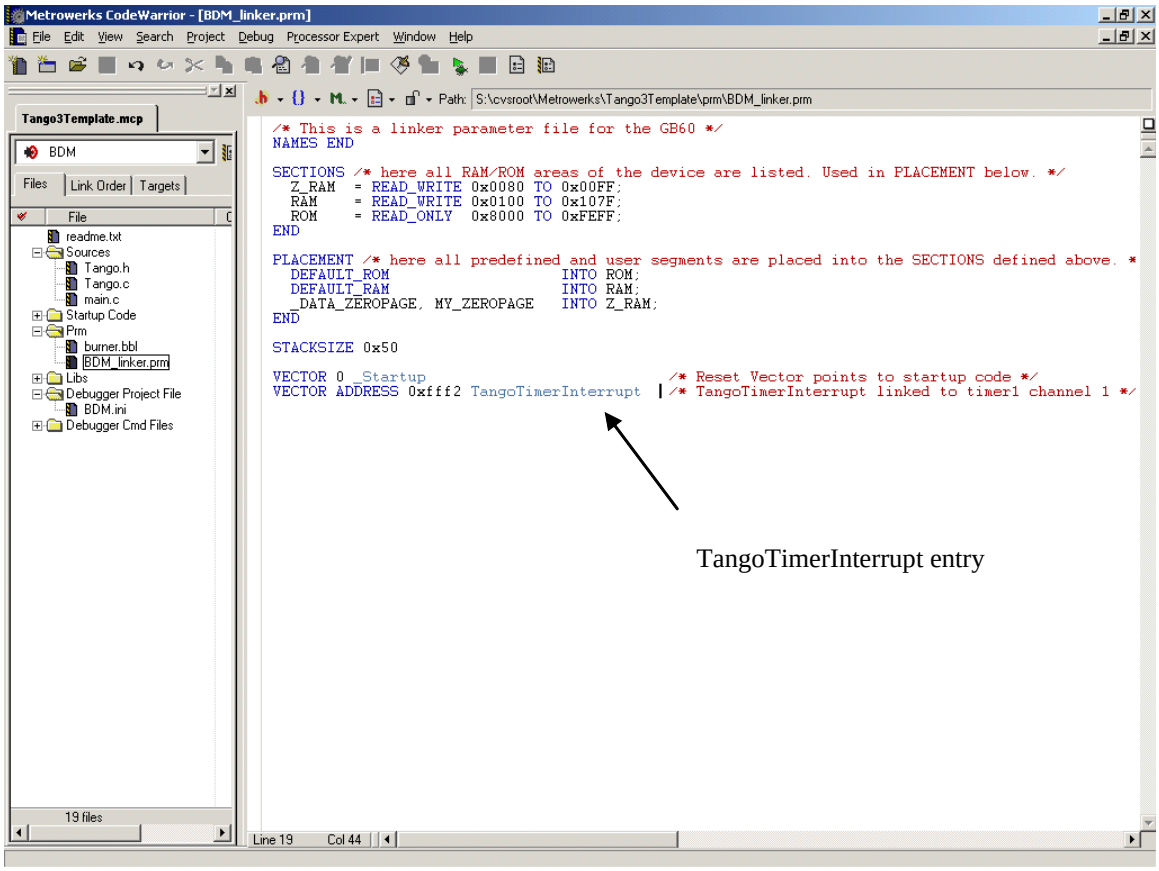


Figure 14. Project Parameter File

```
/*****************************************************************************/
/*            THIS SECTION CONTAINS VALUES YOU MUST DEFINE!                  */
/*                                                                           */
#include <MC9S08GB60.h>                         /* Include peripheral declarations     */

#define TANGO_TIMER_ADDRESS     0x30            /* Location of 1st timer register      */
#define TANGO_TIMER_CHANNEL     1               /* Define which timer channel to use   */
                                                /* Note:timer channels start from 0    */

#define TANGO_MAX_DATA_SIZE 127                 /* Max size of data                    */

                                                /* Set TANGO Mode                      */
#define  TANGO_MODE_VALUE TANGO_OOK             /* TANGO_OOK or TANGO_FSK              */

                                                /* Set timer clock speed in Hz         */
#define TANGO_TIMER_CLOCK_SPEED  2000000

#define TANGO_TIMER_CLOCK_SOURCE    3          /* Use to set clock source for timer   */
                                                /* 1 = Bus clock                       */
                                                /* 2 = XCLK                            */
                                                /* 3 = Ext clock                       */

#define TANGO_CRYSTAL_FREQUENCY  13560000       /* Crystal frequency (in Hz)           */
                                                /* Typical values used                 */
                                                /* RF Output                           */
                                                /* 315MHz  - 98400000                  */
                                                /* 434MHz  - 13560000                  */
                                                /* 868MHz  - 13560000                  */

#define TANGO_TIMER_PRESCALE    1               /* Specify timer prescaler value       */
                                                /* NOTE: If using DATACLK from         */
                                                /* Tango ic, prescaler will be forced  */
                                                /* to 1                                */

#define TANGO_TIMER_DISABLE    1          /* Allows driver to turn off timer after use */
                                           /* Delete this #define if you want timer to */
                                           /* stay on                                   */

                                                /* Set Tango Band                      */
                                                /* TANGO_HIGH_BAND or TANGO_LOW_BAND   */
#define TANGO_BAND_VALUE   TANGO_HIGH_BAND

                                                /* Set Tango data rate in Hz (before   */
#define TANGO_DATA_RATE      1000               /* Manchester encoding)                */


#define TANGO_ENABLE        PTAD_PTAD0              /* Define pin used for enable      */
#define TANGO_ENABLE_DDR    PTADD_PTADD0            /* If hardwired,delete #defines    */

/*****************************************************************************/
/* These may be omitted depending on the hardware setup                      */

#define TANGO_MODE          PTAD_PTAD1              /* Define pin used for mode select */
#define TANGO_MODE_DDR      PTADD_PTADD1            /* If hardwired,delete #defines    */

#define TANGO_BAND          PTAD_PTAD2              /* Define pin for band select      */
#define TANGO_BAND_DDR      PTADD_PTADD2            /* If hardwired,delete #defines    */

//#define TANGO_ENABLE_PA                       /* Define pin used for Power amp enable */
//#define TANGO_ENABE_PA_DDR                    /* If hardwired, delete #defines        */
/*****************************************************************************/
```

Figure 15. Example Tango3.H file

Using the Tango3 Driver in an Application

1. At the start of your application, you must call function 'TangoInitialise()'. This configures the driver and the MCU's timer. Note, to save power, this function does not switch on the Tango3 IC.
2. Before sending commands using Tango3, you must call 'TangoEnable'. This powers up Tango3 (if the ENABLE pin is being used) and starts a 2 ms delay to allow Tango3 to start.
3. Application can now send messages in the two formats described in **Sending Messages With Header Detect on page 5** and **Sending Messages Without Header Detect on page 5**.

To send a message with no header, put the message in the transmit buffer in RAM, then call TangoSendMessageNoHeader().

The application can check the current state of the driver by calling TangoStatus().

Figure 16 shows a simple application that will send a continuous stream of messages using Tango3. Each message contains one data byte and the value of the data byte is incremented.

Figure 15 shows the contents of the Tango.h file for this example.

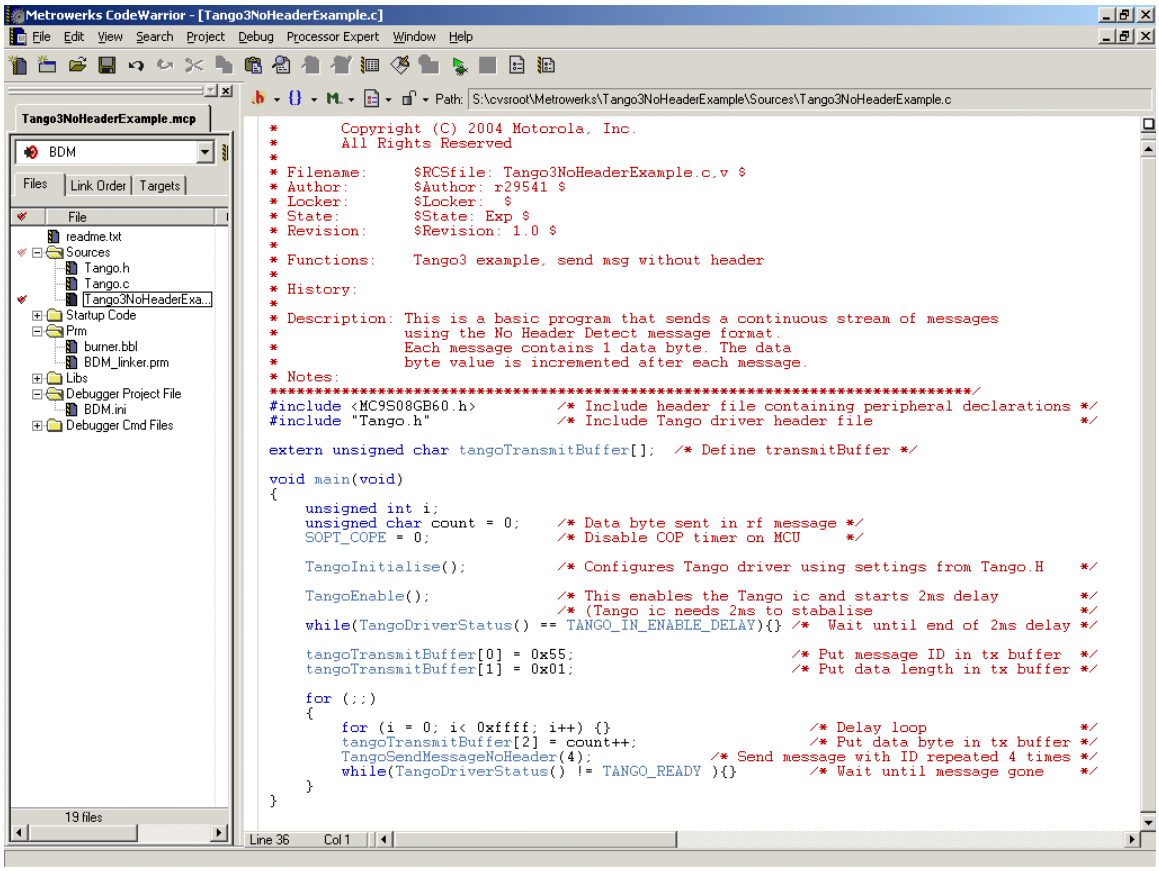


Figure 16. Example Tango3 Application.

Romeo2 Driver

Romeo2 Hardware Connection

Figure 17 shows the interface between the Romeo2 IC and an MCU.

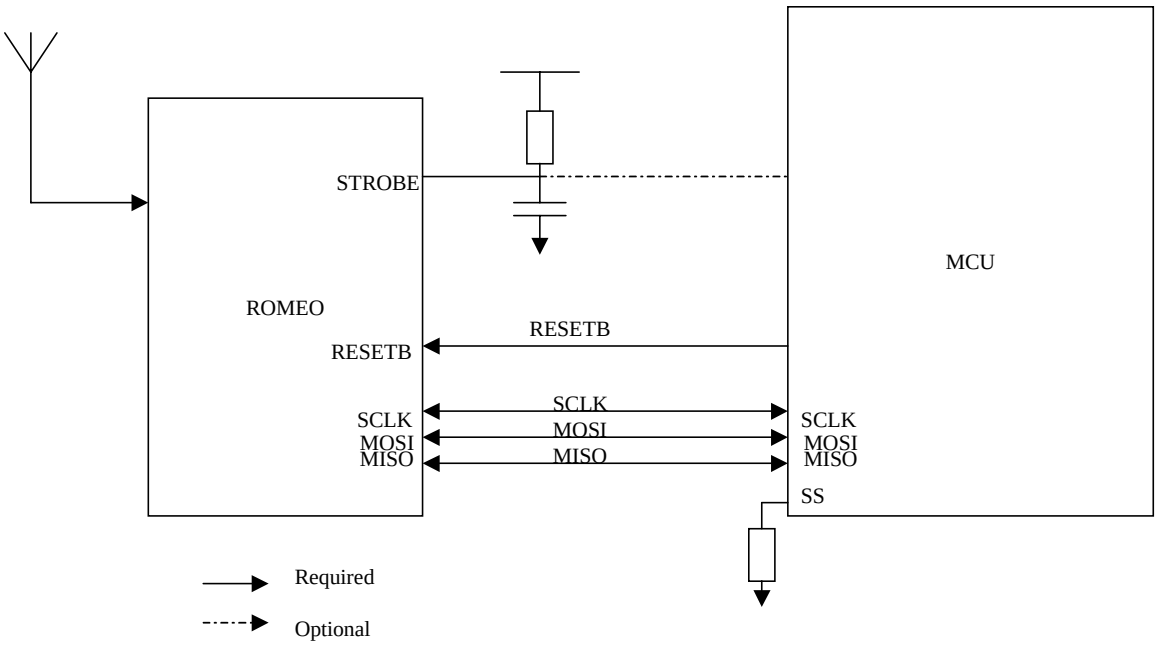


Figure 17. Romeo2 Interface to MCU

MOSI, MISO, SCLK — These are SPI data and clock connections. The SPI link allows the MCU to configure Romeo2, and also allows Romeo2 to pass data received on the RF link to the MCU.

RESETB — This signal controls the state of Romeo2 and the direction of data on the SPI interface. When RESETB is at logic 0, Romeo2 is a slave on the SPI link; the MCU can write or read data to or from Romeo2's internal registers. When RESETB is logic 1, Romeo2 is the master on the SPI bus; it sends data received on RF to the MCU via the SPI.

SS (Slave Select) — This pin on the MCU must be held low when the MCU is configured as an SPI slave. In most systems, this pin will be tied to ground using a pull-down resistor.

Romeo2 Driver Description

This section provides a description of the Romeo2 driver application interface and run-time services.

The Romeo2 driver provides a set of runtime services using C function calls that allow the user to receive messages. The services are listed below.

RomeoInitialise — Configure the Romeo2 driver (must be called when MCU resets)

RomeoEnable — Enables driver (and Romeo2 hardware) for transmission

RomeoDisable — Disables driver (and Romeo2 hardware)

RomeoStatus — Returns current state of driver

RomeoStrobeHigh — Driver sets Romeo2's STROBE pin high

RomeoStrobeLow — Driver sets Romeo2's STROBE pin low

RomeoStrobeTriState — Driver tristates Romeo2's STROBE pin

RomeoChangeConfig — Allows driver to reconfigure Romeo2's internal registers

RomeoSPIRxInt — Provides the driver with a link to the MCU's SPI interface receive interrupt.

The Romeo2 driver defines a receive buffer in RAM. The Romeo2 driver writes complete messages to this buffer after reception from the RF link. The buffer contains the message Length and Data fields and a Buffer Full status flag, as shown in **Figure 18**. The size of the buffer can be programmed by the user, using the ROMEO_MAX_DATA_SIZE parameter in the Romeo.H header file. You should make the buffer large enough to receive the largest message being transferred. See **ROMEO_MAX_DATA_SIZE on page 42** for details.

Note that storage for the ID and Checksum fields is not required. Each Romeo2 device has a fixed ID defined at compile time, so no additional storage is required. The Romeo2 driver calculates the Checksum field for each message internally, and compares it with the actual checksum received. If there is an error, the driver status is updated to ROMEO_CHECKSUM_ERROR.

The application must not read from the receive buffer, nor read the Buffer Full flag, before calling the RomeoStatus service to check if a new valid message is waiting. To do so could result in reading a corrupted message (the driver may store a new message in the buffer, while the application is reading the previous message). After the application has successfully read the message, it must clear the Buffer Full flag.

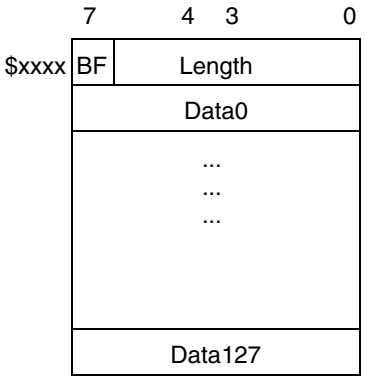


Figure 18. Romeo2 Receive Buffer

If a message is received and the receive buffer is full (BF flag = 1), the last received message will be discarded and the driver status will be set to ROMEO_OVERRUN.

After the driver has been initialized, it can be in one of five states (listed below).

ROMEO_DISABLED — Driver disabled, Romeo2 IC in low power mode.

ROMEO_MSG_READY — Driver enabled, message ready in data buffer.

ROMEO_OVERRUN — Driver enabled, input buffer full, previous message received has been lost.

ROMEO_CHECKSUM_ERROR — Driver enabled, last message received has a checksum error.

ROMEO_NO_MSG — Driver enabled, no messages waiting.

Figure 19 shows a flowchart for configuring the driver to receive messages.

Figure 20 shows the various states the driver will return, when the RomeoStatus service is called.

Internal processing of the driver occurs when the main application calls any of the run-time services, and after reception of data on the SPI interface. Since reception of data is controlled by an SPI interrupt on the MCU, the user must link the RomeoSPIRxInt service to the SPI interrupt. An example of this is given in **Adding the Romeo2 Driver to an Application on page 46**.

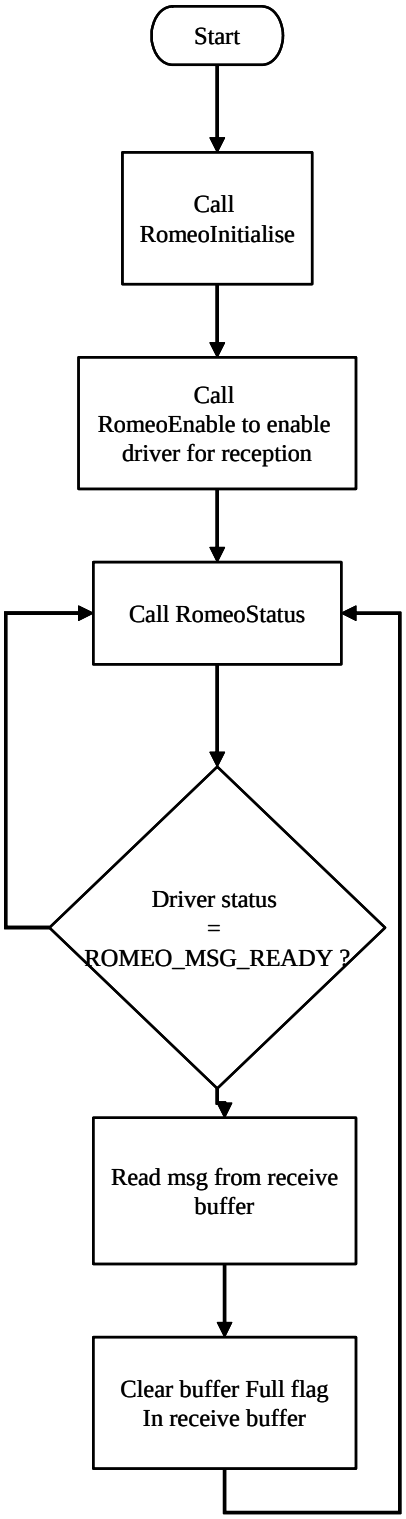


Figure 19. Configuring the Driver to Receive Messages

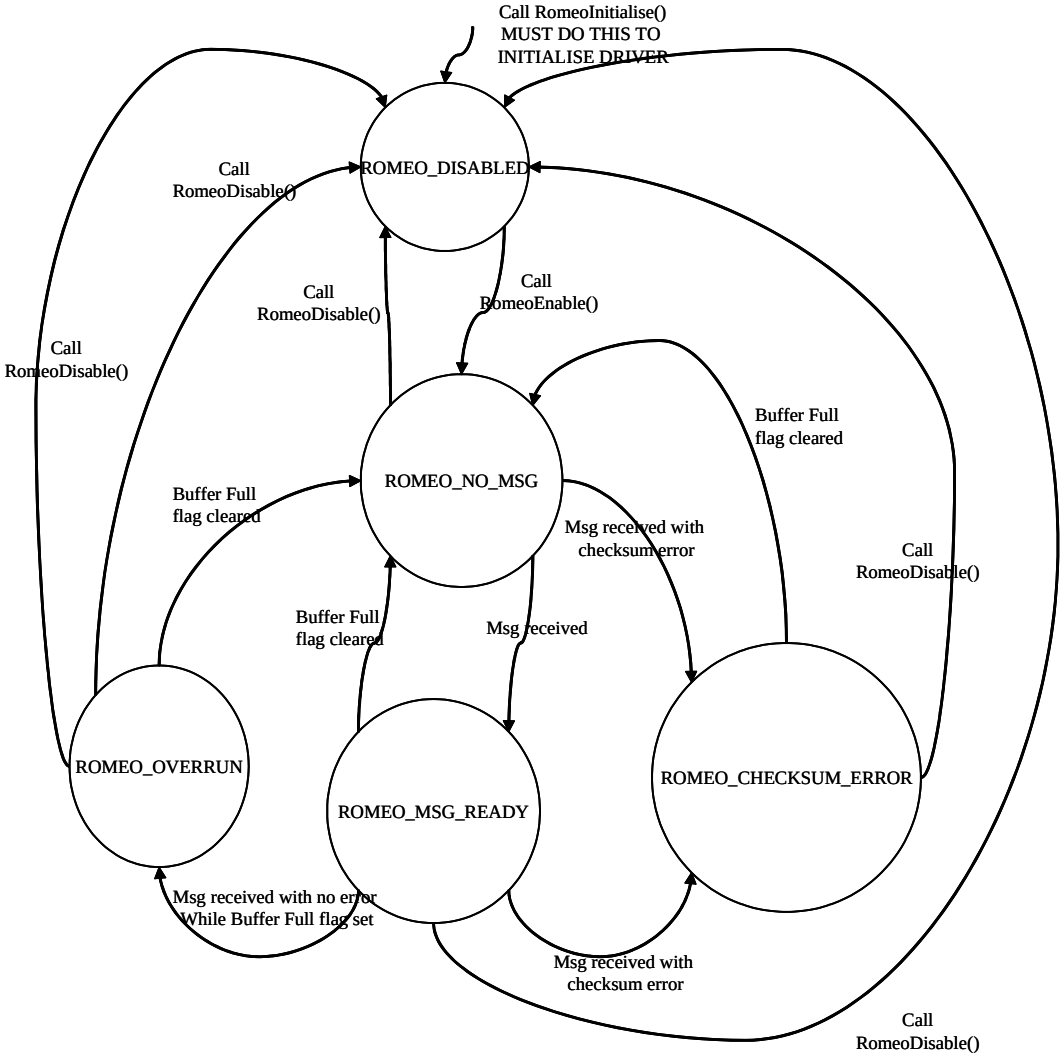


Figure 20. States Returned by the RomeoStatus Service

Romeo2 Driver Services

This section provides descriptions of each service provided by the Romeo2 driver.

RomeoInitialise

Syntax: void RomeoInitialise(void);

Parameters: None

Return: None

Description: The RomeoInitialise service performs initialization of the Romeo2 IC and software driver. It performs the following operations.

- Configures Romeo2 with options defined in Romeo.H file using SPI
- Sets the driver status to ROMEO_DISABLED

Notes: This service should be called before any other Romeo2 driver services. Otherwise, the result of any other Romeo2 driver service and the Romeo2 driver will be unpredictable.

RomeoEnable

Syntax: void RomeoEnable(void);

Parameters: None

Return: None

Description: The RomeoEnable service enables Romeo2 to receive messages. The Strobe line, if under driver control, is taken high to force Romeo2 into RUN mode. Romeo2's SPI interface is configured to make Romeo2 the master, so that it can pass data to the MCU. The driver status is set to ROMEO_NO_MSG.

RomeoDisable

Syntax: void RomeoDisable(void);

Parameters: None

Return: None

Description: The RomeoDisable service disables passing of data from Romeo2 to the MCU and forces the Strobe line low to keep Romeo2 in SLEEP mode. The driver state is set to ROMEO_DISABLED.

RomeoStatus

Syntax: unsigned char RomeoStatus(void);

Parameters: None

Returns:

- ROMEO_DISABLED — driver disabled, Romeo2 IC in low power mode
- ROMEO_MSG_READY — driver enabled, message ready in data buffer
- ROMEO_OVERRUN — driver enabled, input buffer full, previous message received has been lost
- ROMEO_CHECKSUM_ERROR — driver enabled, last message received has a checksum error
- ROMEO_NO_MSG — driver enabled, no messages waiting

Description: The RomeoStatus service returns the current state of the Romeo2 driver.

RomeoStrobeHigh

Syntax: void RomeoStrobeHigh(void);

Parameters: None

Returns: None

Description: The RomeoStrobeHigh service sets the Strobe pin (if under driver control) to logic 1. This service can be called by the application to allow RUN/SLEEP mode cycling of the Romeo IC, to reduce power consumption

RomeoStrobeLow

Syntax: void RomeoStrobeLow(void);

Parameters: None

Returns: None

Description: The RomeoStrobeLow service sets the Strobe pin (if under driver control) to logic 0. This service can be called by the application to allow RUN/SLEEP mode cycling of the Romeo2 IC, to reduce power consumption.

RomeoStrobeTriState

Syntax: void RomeoStrobeTriState(void);

Parameters: None

Returns: None

Description: The RomeoStrobeTriState service sets the Strobe pin (if under driver control) to a high impedance state. This service can be called by the application to allow RUN/SLEEP mode cycling of the Romeo2 IC, to reduce power consumption.

RomeoChangeConfig

Syntax:	void RomeoChangeConfig(unsigned char cr1, unsigned char cr2, unsigned char cr3);
Parameters:	cr1,cr2,cr3
Returns:	None
Description:	The RomeoChangeConfig service allows the application to directly change the contents of the Romeo2 IC's internal 8-bit registers cr1, cr2 and cr3. This gives the user the option to change carrier frequency, switch on/off the strobe function, or change other functions. Please consult the Romeo2 IC datasheet for a full description of the contents of these registers.

RomeoSPIRxInt

Syntax:	interrupt void RomeoSPIRxInt(void);
Parameters:	None
Returns:	None
Description:	This function is called by the interrupt vector of the SPI interface used to communicate with the Romeo2 IC. In the CodeWarrior parameter file, the SPI interrupt vector must be directed to this function. This function MUST be included to ensure proper operation of the software driver.

Romeo2 Driver Configuration

The Romeo2 driver has a static configuration at compile time. Its configuration cannot be changed during run time. The driver configuration is defined in a header file 'Romeo.h'. Configuration options are available for:

- Message format
- Message data rate
- Message modulation format (OOK or FSK)
- Carrier frequency
- Strobe oscillator function
- MCU resources

These configuration options are set using a number of #define statements in the Romeo.h header file. Using these #defines, the driver can be configured to run on any HC08 MCU with an SPI interface.

When starting a new project using the Romeo2 driver, you should place files ‘Romeo.H’ and ‘Romeo.C’ in the project directory and a #include ‘Romeo.H’ statement in the main application file.

The Romeo.H file contains a number of #define statements that must be configured to ensure correct operation of the driver. These are described below:

ROMEO_SPI_ADDRESS

Description: This defines the start address of the SPI control registers in the MCU's memory map.

Values: Integer in range 0x0000–0xffff

Example:

```
#define ROMEO_SPI_ADDRESS    0x10          /* Address varies from MCU to MCU  */
```

ROMEO_MAX_DATA_SIZE

Description: This defines the maximum number of data bytes that can be transferred. This value is used to calculate the size of the message receive buffer (receive buffer will be ROMEO_MAX_DATA_SIZE + 1 byte.)

Values: Number in range 0–127

Example:

```
#define ROMEO_MAX_DATA_SIZE     8       /* Max size of data field = 8 bytes   */
```

ROMEO_RESET

Description: This defines the I/O pin used to control Romeo2's RESET pin.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define ROMEO_RESET          PTA_PTA0        /* Port A pin 0 used for RESET   */
```

ROMEO_RESET_DDR

Description: This defines the data direction bit for the I/O pin used to control Romeo2's RESET pin.

Values: Any I/O pin configurable as an output can be used. Use the naming convention specified in the CodeWarrior header files.

Example:

```
#define ROMEO_RESET_DDR          PTA_PTA0        /* DDRA for Port A pin 0    */
```

ROMEO_MODE_VALUE

Description: This defines the modulation type used in RF transmissions - ON/OFF Keying (ROMEO_OOK) or Frequency Shift Keying (ROMEO_FSK)

Values: ROMEO_OOK = OOK modulation
ROMEO_FSK = FSK modulation

Example:

```
#define ROMEO_MODE_VALUE    ROMEO_OOK       /*  OOK modulation */
```

ROMEO_BAND_VALUE

Description: This defines if Romeo2 is used in high band or low band configuration.

Values: ROMEO_HIGH_BAND = high band selected
ROMEO_LOW_BAND = low band selected

Example:

```
#define ROMEO_BAND_VALUE    ROMEO_HIGH_BAND    /* High band selected     */
```

ROMEO_SOE_VALUE

Description: This defines if the Strobe oscillator is enabled on Romeo2

Values: 0 = disabled
1 = enabled

Example:

```
#define ROMEO_SOE_VALUE     1              /* Strobe oscillator enabled*/
```

ROMEO_HE_VALUE

Description: This defines if Romeo2 uses the header detect messaging format.

Values: 0 = no header byte present in messages
1 = header detect messaging used

Example:

```
#define ROMEO_HE_VALUE      0              /* 0 = No header word used*/
```

ROMEO_ID_VALUE

Description: This defines the ID word used for this particular Romeo2 IC.

Values: Integer in range 0 - 0xff that does not contain binary sequence 0110

Example:

```
#define ROMEO_ID_VALUE      0x55           /* ID word set to 0x55      */
```

ROMEO_SPI_CLOCK_SPEED

Description: This defines the speed of the SPI clock.

Values: Integer in range 0 - 20000000

Example:

```
#define ROMEO_SPI_CLOCK_SPEED   8000000/* SPI clock is 8MHz */
```

ROMEO_SR_VALUE

Description: This defines the ratio SLEEP time over RUN time for the strobe oscillator.

Values: 0 - strobe ratio = 3
1 - strobe ratio = 7
2 - strobe ratio = 15
3 - strobe ratio = 31

Example:

```
#define ROMEO_SR_VALUE      1                   /* Sleep time is 7 x RUN time */
```

ROMEO_DR_VALUE

Description: This defines the data rate of received messages before Manchester encoding

Values: 0 = 1.0 - 1.4kbaud
1 = 2.0 - 2.7kbaud
2 = 4.0 - 5.3kbaud
3 = 8.6 - 10.6kbaud

Example:

```
#define ROMEO_DR_VALUE       0              /* Datt in range 1.0 - 1.4 kbaud */
```

ROMEO_MG_VALUE

Description: This defines the gain of Romeo2's mixer stage.

Values: 0 = Normal gain
1 = -17dB (typical)

Example:

```
#define ROMEO_MG_VALUE      0              /* Mixer gain is norma          */
```

ROMEO_MS_VALUE

Description: This #define switches the position of the MIXOUT pin.

Values: 0 - MIXOUT at mixer output
1 - MIXOUT at IF input

Example:

```
#define ROMEO_MS_VALUE      0             /* MIXOUT pin to mixer output     */
```

ROMEO_PG_VALUE

Description: This define sets the gain of the phase comparator.

Values: 0 - high gain mode
1 - low gain mode

Example:

```
#define ROMEO_PG_VALUE      1       /* Phase comparator set to low gain mode */
```

Adding the Romeo2 Driver to an Application

To add the Romeo2 driver to an application:

1. Add Romeo.h and Romeo.c files to project (in CodeWarrior, right click on sources folder, then add files).
2. Add line #include ‘Romeo.h’ to main application program file.
3. Add line 'extern unsigned char romeoReceiveBuffer[];' to main application program file.
4. Decide which I/O pins in your application will control Romeo2 functions. Modify the Romeo.H file to link these pins to Romeo2.

5. Modify Romeo.H file to define timer speed, and other parameters.

The files are now added to the project

Figure 21 is a screenshot of a CodeWarrior application template, with Romeo2 files included, and showing the main application program file with correct entries added.

Figure 22 is a screenshot of the project parameter file showing how to include the RomeoSPIRxInt function call.

Figure 23 shows an example Romeo.H header file. This has been configured for use with an MC68HC908GZ60 MCU Romeo2 is configured for 434 MHz operation with a data rate of 1–1.4 kbps.

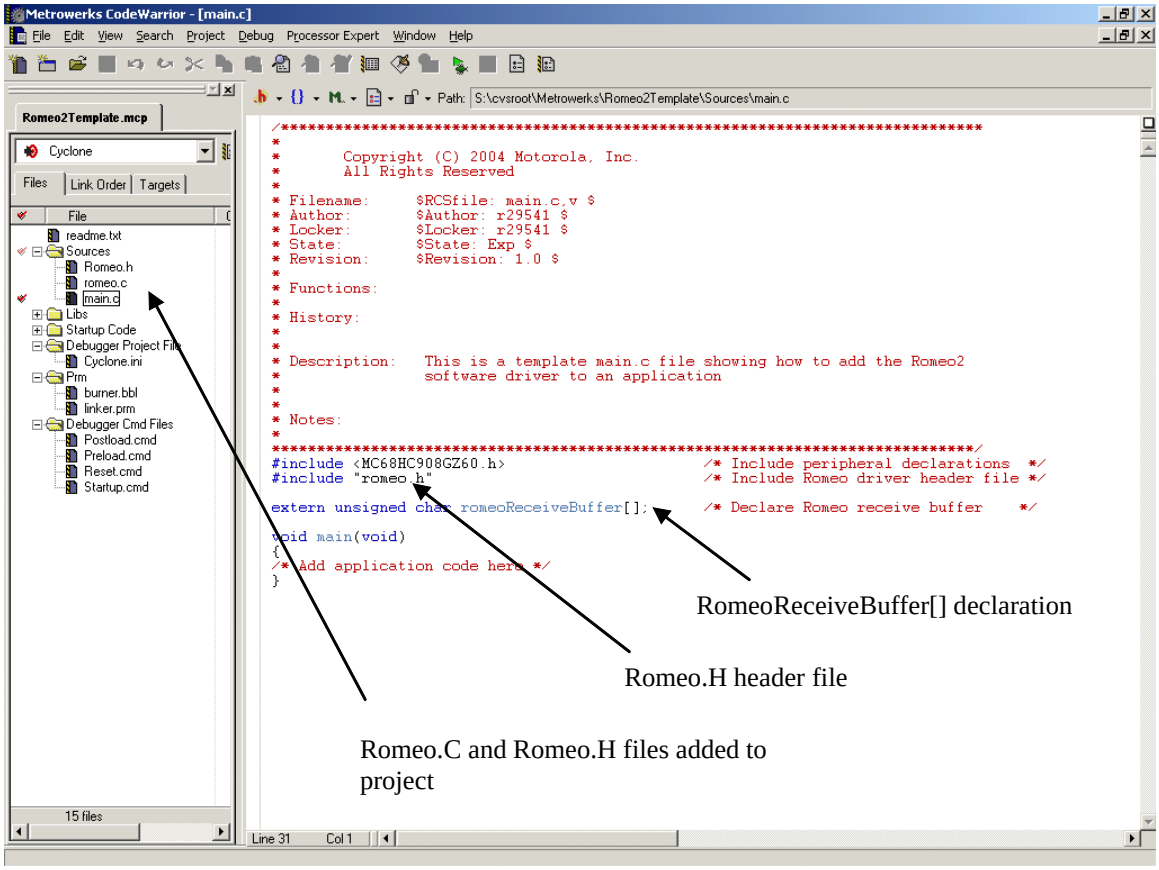


Figure 21. Romeo2 Application Template

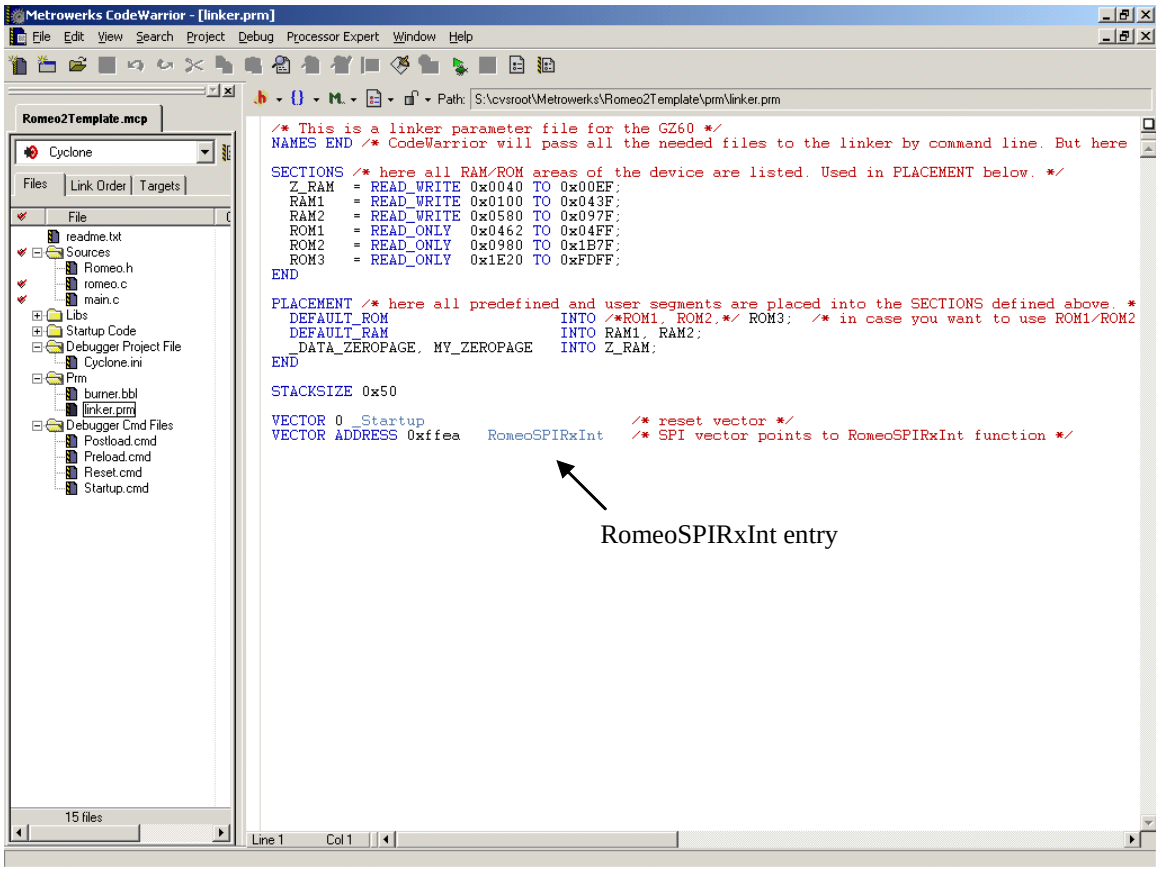


Figure 22. Project Parameter File

```
/*****************************************************************************/
/*           THIS SECTION CONTAINS VALUES YOU MUST DEFINE!                   */
/*****************************************************************************/
#include <MC68HC908GZ60.h>                          /* include peripheral declarations */

/* Specify start adress of SPI registers                                     */
#define ROMEO_SPI_ADDRESS   0x10                    /* Address varies from mcu to mcu  */

/* Set length of data field in receive data buffers                          */
#define ROMEO_MAX_DATA_SIZE 8                       /* Max length of data field in msg */

/* Set Romeo reset pin                                                       */
#define ROMEO_RESET         PTG_PTG0                /* Define pin used for Reset       */
#define ROMEO_RESET_DDR     DDRG_DDRG0

/* Set Romeo mode                                                            */
#define ROMEO_MODE_VALUE    ROMEO_OOK               /* ROMEO_OOK = OOK reception       */
                                                    /* ROMEO_KSF = FSK reception       */
/* Set Romeo band                                                            */
#define ROMEO_BAND_VALUE    1                       /* 0 = lower band                  */
                                                    /* 1 = higher band                 */
/* Enable/disable Strobe osc                                                 */
#define ROMEO_SOE_VALUE     1                       /* 0 = strobe oscillator disabled  */
                                                    /* 1 = strobe oscillator enabled   */

/* Header word present select                                                */
#define ROMEO_HE_VALUE      1                       /* 0 = No header word used         */
                                                    /* 1 = Header word present         */

/* Define ID word value                                                      */
#define ROMEO_ID_VALUE      0x55                    /* ID word recognised by Romeo     */

/* SPI clock speed                                                           */
#define ROMEO_SPI_CLOCK_SPEED   8000000

/* Strobe Ratio value                                                        */
#define ROMEO_SR_VALUE      1                     /* 0 = strobe ratio  3               */
                                                  /* 1 = strobe ratio  7               */
                                                  /* 2 = strobe ratio  15              */
                                                  /* 3 = strobe ratio  31              */
/* Data rate                                                                 */
#define ROMEO_DR_VALUE      0                     /* 0 = 1.0 - 1.4kbaud                */
                                                  /* 1 = 2.0 - 2.7kbaud                */
                                                  /* 2 = 4.0 - 5.3kbaud                */
                                                  /* 3 = 8.6 - 10.6kbaud               */
/* Mixer gain                                                                */
#define ROMEO_MG_VALUE      0                     /* 0 = Normal                        */
                                                  /* 1 = -17dB (typical)               */
/* MS switch                                                                 */
#define ROMEO_MS_VALUE      0                     /* 0 = to mixer output               */
                                                  /* 1 = to IF input                   */

/* Phase comparator gain                                                     */
#define ROMEO_PG_VALUE      1                     /* 0 = high gain mode                */
                                                  /* 1 = low gain mode                 */

/*****************************************************************************/
/*           These may be omitted depending on hardware setup                */
/*****************************************************************************/
#define ROMEO_STROBE        PTG_PTG1          /* #defines for STROBE pin               */
#define ROMEO_STROBE_DDR    DDRG_DDRG1        /* If hardwired,delete #defines          */

#define ROMEO_AGC           PTG_PTG3          /* #defines for AGC pin                  */
#define ROMEO_AGC_DDR       DDRG_DDRG3        /* If hardwired,delete #defines          */

#define ROMEO_AGC_VALUE     1               /* 1 -> slow, OOK                          */
                                            /* 0 -> fast, FSK                          */

/*These are required for use with Motorola's rf modules                     */
#define ROMEO_ENABLELNA     PTG_PTG2          /* #defines for LNA pin                  */
#define ROMEO_ENABLELNA_DDR DDRG_DDRG2        /* If hardwired,delete #defines          */
```

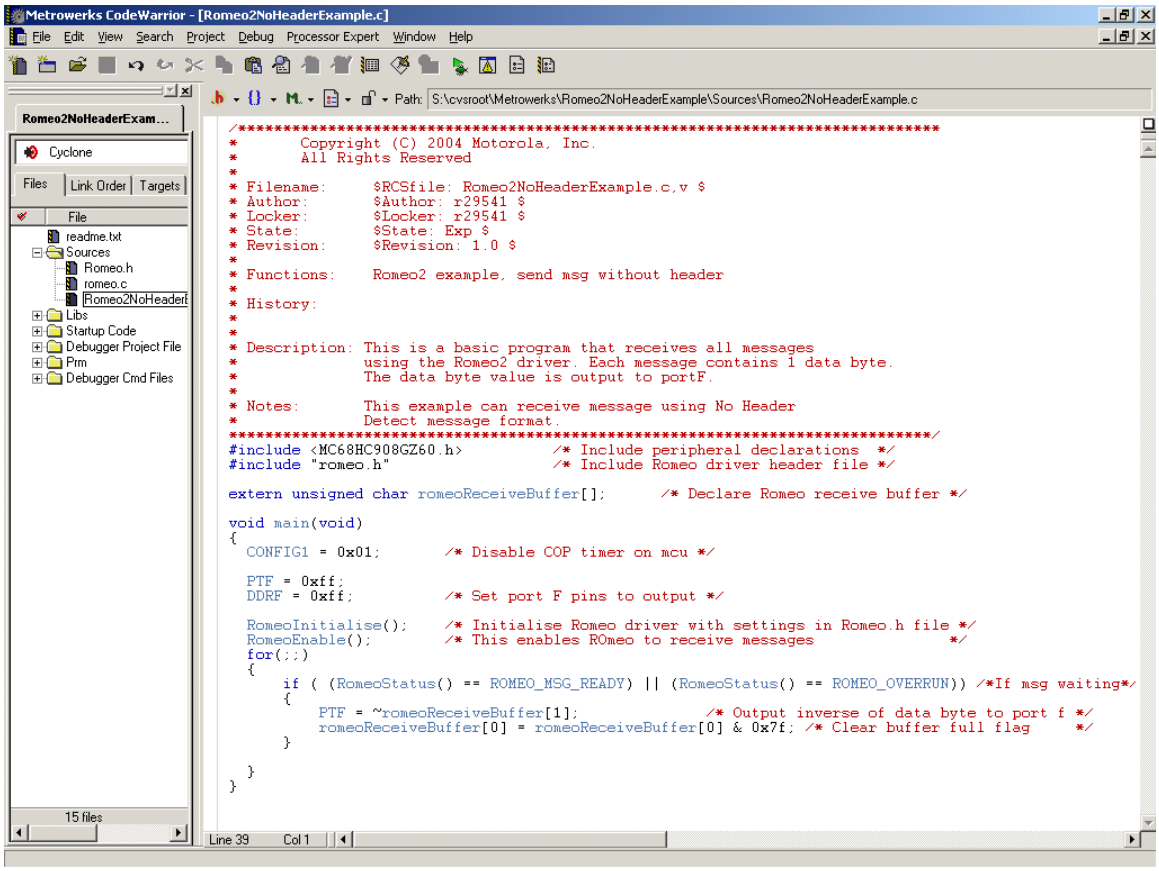
Figure 23. Example Romeo.h File

Using the Romeo2 Driver in an Application

1. The application must first call RomeoInitialise() to configure the driver correctly.
2. The application must then call RomeoEnable() to enable the Romeo2 IC to receive messages and the driver to process them.
3. After RomeoEnable() has been called, the application should poll the status of the driver using RomeoStatus(). If the status is ROMEO_MSG_READY, or ROMEO_OVERRUN, a message is waiting in romeoReceiveBuffer.
4. After a message has been read from the receive buffer, the Buffer Full flag in the receive buffer (bit 8 in byte 0 of buffer) should be cleared to indicate the receive buffer is now available for new messages.

Figure 24 shows a simple example using the Romeo2 driver that receives all messages.

Figure 23 shows the contents of the Romeo.h file for this example.




```
/*****************************************************************************
*       Copyright (C) 2004 Motorola, Inc.
*       All Rights Reserved
*
* Filename:     $RCSfile: Romeo2NoHeaderExample.c,v $
* Author:       $Author: r29541 $
* Locker:       $Locker: r29541 $
* State:        $State: Exp $
* Revision:     $Revision: 1.0 $
*
* Functions:    Romeo2 example, send msg without header
*
* History:
*
*
* Description: This is a basic program that receives all messages
*              using the Romeo2 driver. Each message contains 1 data byte.
*              The data byte value is output to portF.
*
* Notes:       This example can receive message using No Header
*              Detect message format.
*****************************************************************************/
#include <MC68HC908GZ60.h>          /* Include peripheral declarations  */
#include "romeo.h"                  /* Include Romeo driver header file */

extern unsigned char romeoReceiveBuffer[];      /* Declare Romeo receive buffer */

void main(void)
{
  CONFIG1 = 0x01;       /* Disable COP timer on mcu */

  PTF = 0xff;
  DDRF = 0xff;          /* Set port F pins to output */

  RomeoInitialise();    /* Initialise Romeo driver with settings in Romeo.h file */
  RomeoEnable();        /* This enables ROmeo to receive messages               */
  for(;;)
  {
      if ( (RomeoStatus() == ROMEO_MSG_READY) || (RomeoStatus() == ROMEO_OVERRUN)) /*If msg waiting*/
      {
          PTF = ~romeoReceiveBuffer[1];             /* Output inverse of data byte to port f */
          romeoReceiveBuffer[0] = romeoReceiveBuffer[0] & 0x7f; /* Clear buffer full flag     */
      }

  }
}
```

Figure 24. Example Romeo2 Application

Trademarks

- Motorola and the Motorola logo are registered trademarks of Motorola, Inc.
- CodeWarrior® is a registered trademark of MetroWerks, a wholly owned subsidiary of Motorola, Inc.

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:
Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

ASIA/PACIFIC:
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors

MOTOROLA

Freescale Semiconductor, Inc.