Freescale Semiconductor
Application Note

Document Number: AN3333
Rev. 0, 11/2006

XGATE Library: Signal Gateway

Implementing CAN and LIN Signal Level Gateway

by: Daniel Malik
MCD Applications
East Kilbride, Scotland

1 Introduction

The XGATE signal gateway package is a collection of header and source files in the C programming language, enabling you to create a CAN and LIN signal level automotive body gateway. The gateway structure detailed in this application note is flexible enough to allow extensions and/or modifications to satisfy a particular environment's requirements.

2 Signal Gateway Theory

No industry-wide standard currently exists for automotive body gateways. Each vehicle manufacturer maintains his or her own set of evolving specifications of the traffic a gateway handles. This section outlines one set of specifications. These specifications are not based on requirements set by any specific vehicle manufacturer. To gain insight into the XGATE performance, this set represents meaningful basic body gateway requirements enabling a real-world gateway to be implemented.

Contents

1 Introduction 1
2 Signal Gateway Theory 1
2.1 The Purpose of Gateways 2
2.2 Data Representation 3
2.3 Gateway Data Flow 4
3 Gateway Implementation 7
3.1 Data Structures 7
3.2 Algorithms 14
3.3 Limitations of This Gateway Implementation 23
4 Performance Analysis 23
4.1 XGATE Load 23
4.2 Required Memory Size 32
5 Generating the Descriptors 34
5.1 Generated Files 34
5.2 Node Data Worksheet 35
5.3 Rx Frames Worksheet 35
5.4 Tx Frame Worksheet 37
5.5 Generating the Source Files 38
6 References 38

freescale™ semiconductor

2.1 The Purpose of Gateways

In the past decade, the automotive industry has seen increased demand for safety features, increased pressure to reduce environmental effects caused by de-processing vehicles, and increased expectations for modern technology (such as navigation systems or mobile communications) to become standard even in lower-end models.

These changes are partially caused and partially enabled by the decreasing cost of semiconductor solutions offered to automotive manufacturers. Key automotive technologies have developed from networked systems to distributed systems.

The different in-vehicle systems use different networking technologies; however, the traditional controller area network (CAN) developed by Robert Bosch GmbH in 1986 is still at the heart of many. The need for cost-optimized, low-speed buses has led to the development of a local interconnect network (LIN) typically used for seat management and door sub-systems (locks, windows, mirrors, etc.).

Independent buses (as opposed to a single bus interconnecting all systems in the whole vehicle) are used in individual vehicle systems for several reasons:

- Application requirements may result in different communication speeds for different systems (500 kbps for powertrain vs. 125 kbps for dashboard).
- The physical layer of the buses may not be compatible (LIN vs. CAN, low-speed CAN vs. high-speed CAN, etc.).
- Physical separation of the buses may be required for security purposes.
- Systems sharing little information may use physically separate buses to lower the amount of unnecessary traffic.

The gateway application's purpose is to transfer information among the vehicle's different buses to ensure all systems receive the required information in a timely manner. For example, the dashboard is designed to display (among other information) the current temperature of the engine coolant. It is the gateway's responsibility to ensure the temperature information present on the powertrain bus is re-transmitted on the body bus (see Figure 1).

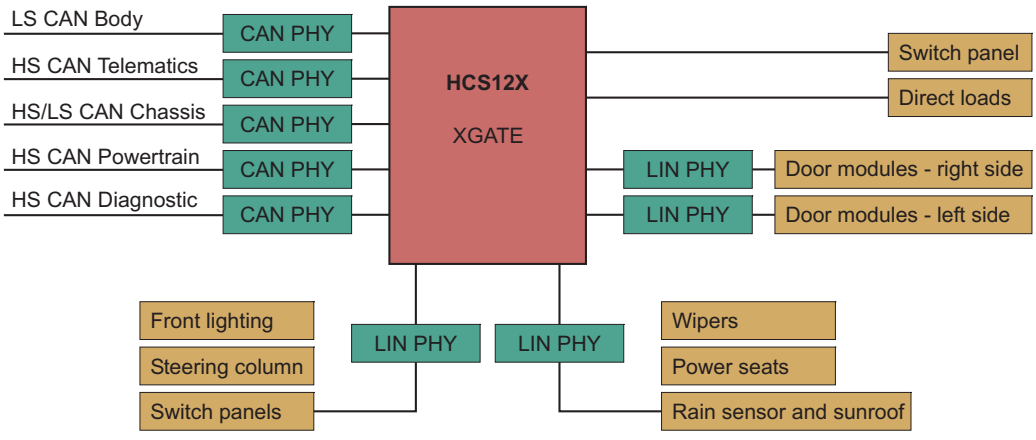

Figure 1. Gateway Application Example

2.2 Data Representation

To assess the task of the gateway application, the user must understand how information is represented on the different buses.

2.2.1 Message Gatewaying

The common property of LIN and CAN frames is that they can transport up to eight data bytes. The simplest task for the gateway is to receive a frame on one bus then re-send it on another bus in its entirety. This process is usually referred to as message gatewaying or as 1:1 frame forwarding and is typically used for diagnostic purposes. In diagnostic mode, the gateway extracts a set of frames from the different buses to which it is connected. These frames are then re-transmitted on the diagnostic bus to which the garage diagnostic equipment is connected. The gateway also receives frames from the diagnostic equipment and forwards them to the different vehicle systems (for adjustment of engine parameters, firmware upgrade, etc.).

2.2.2 Signals

Message gatewaying is efficient; however, it can be used only if the destination application can process the original frame without any transformation.

Another aspect is the bus load. A piece of information is rarely transmitted alone in its own frame. It is common to concatenate more data into a single frame. For example, the engine coolant temperature, the oil temperature, and the intake-air temperature might be concatenated together and transmitted in a single frame on the powertrain bus. If the destination application needs only one byte of information from an 8-byte long frame, the remaining seven bytes would be transmitted unnecessarily and contribute only to dummy load of the destination bus. The pieces of information transported in the frames are called signals. A signal can be between one bit (binary on/off information) and 64 bits (a full 8-byte frame) long.

2.2.2.1 Signal Copy

The bulk task for the gateway is to extract the required signals from the received frames, re-pack them to form new frames, and transmit these on their destination bus. This process is outlined in Figure 2.

Figure 2 shows the signal bits (in blue) received in a 4-byte source (Rx) frame. The gateway's task is to transport this signal to its designated position within the 3-byte destination (Tx) frame in preparation for the destination-frame transmission.

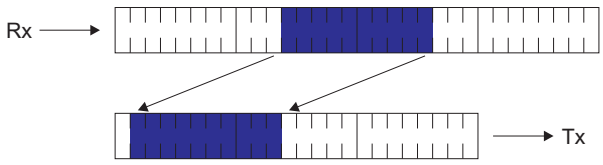

Figure 2. Copy of a Signal from Source Frame to Destination Frame

2.2.2.2 Big and Little Endian

Figure 2 omits any bit and byte numbers within the Rx and Tx frames. Microprocessor manufacturers have been divided into two major groups, depending on how their products access multi-byte entities in their on-chip and off-chip memories. Microprocessors storing the most significant byte on the lowest address are known as big endian architectures. Conversely, the little endian microprocessors store the most significant byte on the highest address.

The HCS12X microcontroller family belongs to the big endian group. The CPU or XGATE can work with data stored in the little endian format; however, care is required in such a case.

The difference between little and big endian architectures is depicted in Figure 3, which shows an 11-bit long signal stored in two consecutive bytes in memory.

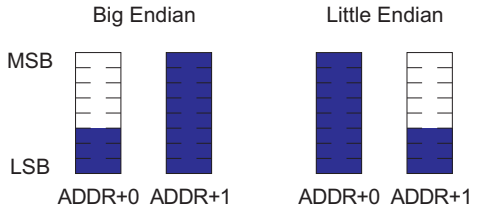

Figure 3. The Difference Between Signal Memory Layout in Big and Little Endian

The difference between the two data organizations is apparent when we attempt to interpret the data not as individual bytes but as a stream of bits. When bytes are depicted horizontally, it is common to keep the least significant bit (LSB) on the right and the most significant bit (MSB) on the left. However, this only works for the big endian data organization (Figure 4). To keep the bits belonging to the signal together, the order of bits in bytes must be reversed for the little endian organization. The consequences of these bit-ordering requirements are discussed further in Section 3.1.1, “Bit and Byte Numbering in Buffers.”

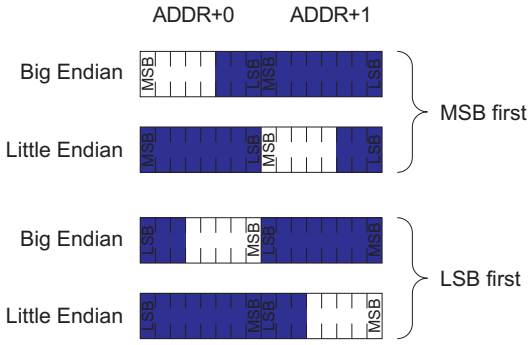

Figure 4. Interpreting Big and Little Endian Data as Streams of Bits

2.3 Gateway Data Flow

This section describes the data flow through the gateway. It outlines how the gateway application should react to incoming signals and when and how the outgoing signals should be transmitted.

2.3.1 Reception

Whenever any interface (node) receives a data frame, the gateway must check whether the frame should be processed (there might be traffic on the buses irrelevant to the gateway process).

In case the incoming frame will be further processed, the gateway must look up the signal information. This includes positions of the individual signals contained in the received frame and their sizes.

After the positions and sizes of signals in the incoming frame are identified, the gateway can start the signal copy process. This involves copying the signals from the Rx buffer to the destination Tx buffers in preparation of signal transmissions. Each signal can have more than one destination (for example, a signal containing the vehicle speed must be forwarded to the dashboard for indication as well as the audio system for volume adjustment purposes).

The gateway does not keep a copy of the received frames. After all the signals from the Rx frame are copied into the Tx buffers, the Rx frame contents are discarded.

Under normal circumstances, a majority of the frames are present on the buses periodically. If a frame ceases to be received, it is usually a sign of a system overload or failure. The gateway application must be capable of tracking the reception frequency for each frame. There are two kinds of timeouts defined for the Rx frames. After the first (shorter) timeout, the gateway recognizes that a reception of a frame is late and that signal values stored in the Tx buffers are out-of-date and potentially incorrect. To indicate this, the gateway can set a flag (a one-bit signal) in one of the Tx buffers. After the second (longer) timeout without reception of a frame, the gateway considers the frame source to have major difficulties in meeting its requirements. The action taken depends on the failing system's function and cannot be generalized (that is, a failure of the entertainment system is inconvenient, but a failure of the airbag system is potentially life threatening).

Figure 5 depicts the reception process.

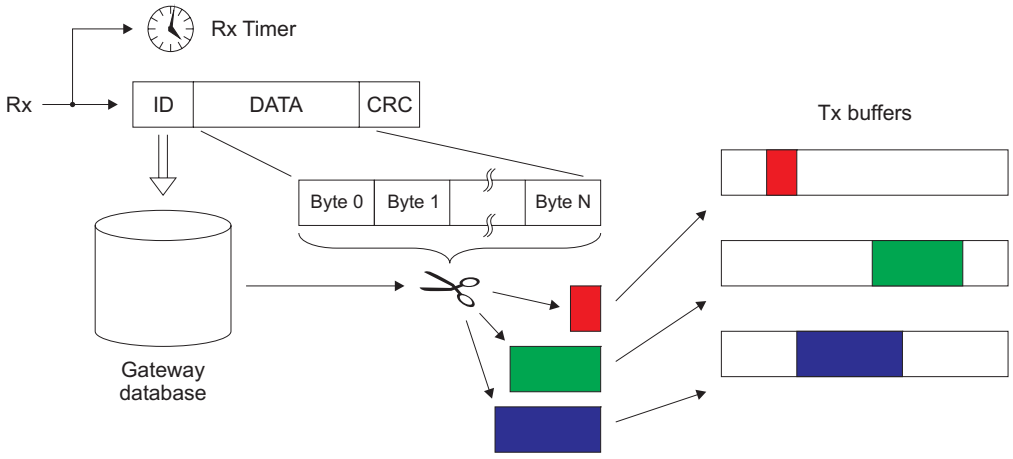

Figure 5. Gateway Reception Process

2.3.2 Transmission

The gateway transmits a majority of Tx frames periodically. The transmission is independent of the reception process, and the gateway keeps an up-to-date copy of all transmitted frames. The gateway keeps a timer for each Tx frame and transmits the frame when the timer expires (and reloads the timer automatically with the specified period). Figure 6 outlines this process.

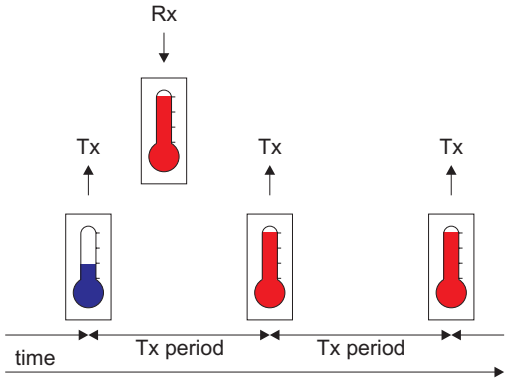

Figure 6. Periodic Frame Transmission

In certain cases, the signals should be transmitted without waiting for the next periodic transmission. The gateway offers the possibility to transmit a Tx frame as soon as any of its signals are received and processed (Figure 7).

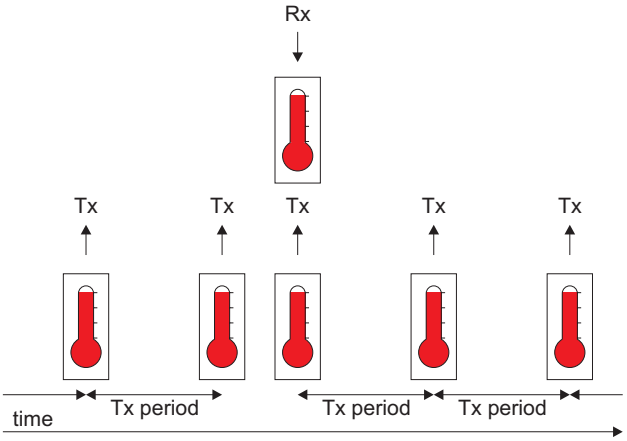

Figure 7. Periodic Frame Transmission with Immediate Transmission upon Signal Reception

In certain cases, it is important to transmit the signals as soon as possible without waiting for the next periodic transmission only if the value of the signal has changed. The gateway supports an option to transmit the Tx frame after any of its signals changes value (Figure 8).

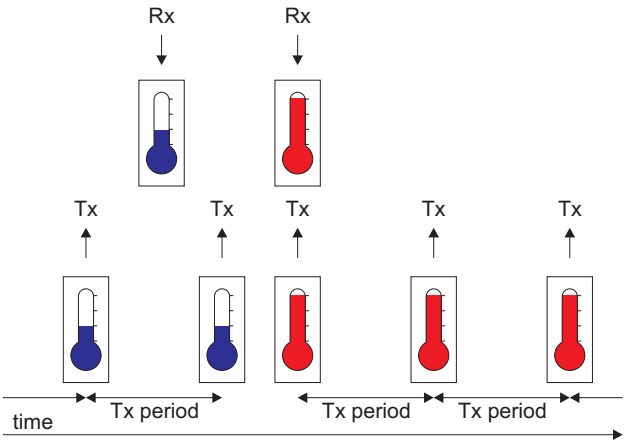

Figure 8. Periodic Frame Transmission with Immediate Transmission upon Signal Value Change

Frame transmission can also be set to happen only after signal reception (with or without value change). Frames transmitted this way are called sporadic frames.

The gateway application maintains a transmission queue for each node to resolve simultaneous transmission of more than one frame.

2.3.3 LIN Considerations

The gateway application is expected to act as a LIN master: it must initiate all transfers through the LIN nodes. The behavior of the Rx timers normally used for timeout processing is changed in case of LIN nodes because they are used to trigger the reception of frames from LIN slaves instead.

3 Gateway Implementation

This section describes a particular implementation of the gateway application for the XGATE co-processor. The implementation makes some limited use of the special XGATE properties; however, a majority of the code is written in ANSI C language.

3.1 Data Structures

This section describes the data structures used in the gateway application. The gateway behavior is fully described by these data structures' content (the gateway database). The same set of universal algorithms processes all incoming and outgoing signals, and the gateway database describes how the different signals are dealt with. None of the signal-specific behaviors are hard-coded in the algorithms. Figure 9 is an overview of the data structures the gateway application uses.

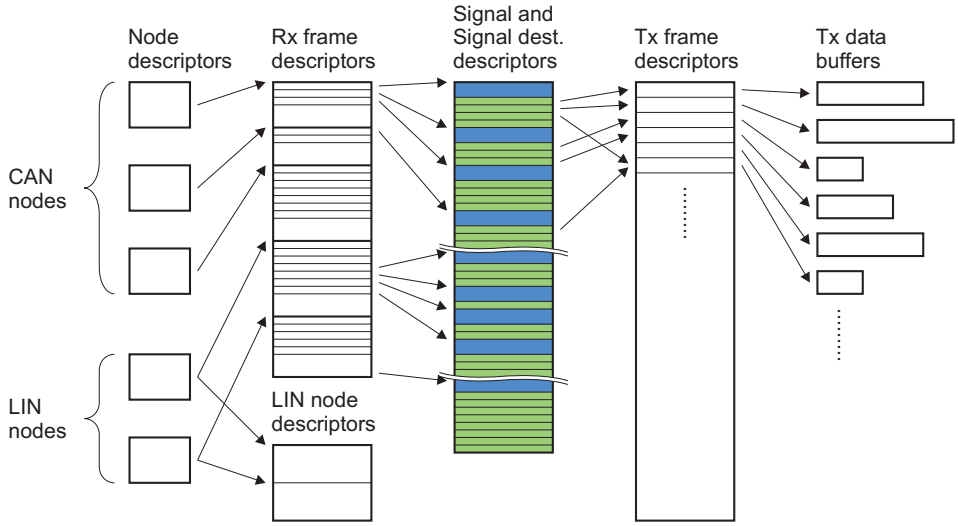

Figure 9. Overview of Gateway Data Structures

3.1.1 Bit and Byte Numbering in Buffers

As described in Section 2.2.2.2, "Big and Little Endian," care is required regarding big and little endian data storage properties. The gateway application can efficiently deal with both encodings; however, the two encodings cannot be mixed. In the implementation, the choice must be made at the time of compilation. The big endian data organization is selected by default; the little endian data storage can be selected by the defining symbol LITTLE_ENDIAN.

3.1.1.1 Big Endian

Figure 10 shows the bit and byte numbering scheme used for frames in big endian format. The bytes are in the usual format: MSB on the left, LSB on the right. However, bit number 0 is the MSB of byte 0, and bit number 63 is the LSB of byte 7. Frames with bytes fewer than eight are aligned to ADDR+0 (for example, a 3-byte frame contains bytes 0, 1, and 2).

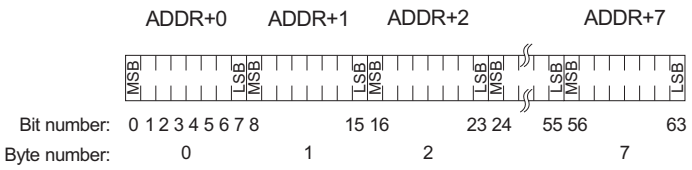

Figure 10. Bit and Byte Numbering Scheme for Big Endian Data Storage

3.1.1.2 Little Endian

Figure 11 shows the bit and byte numbering scheme used for frames in little endian format. The byte numbering is the same as for big endian format, but the order of bits in bytes is different: bit number 0 is the LSB of byte 0, and bit number 63 is the MSB of byte 7.

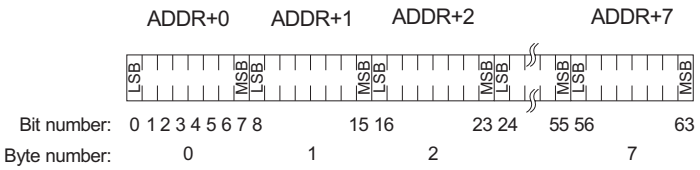

Figure 11. Bit and Byte Numbering Scheme for Little Endian Data Storage

3.1.2 Timer Structure

The timer structure (*tTimer*) is used for all Rx and Tx timers in the gateway application. It contains an 8-bit down-counter and an 8-bit reload value. The down-counter is reloaded when it is decremented down to a value of zero. The frequency of decrementing the down-counter is defined outside of this structure. The timer structure occupies two bytes of memory space. Timer structure:

```
typedef struct {
    unsigned char load_value;
    unsigned char counter;
} tTimer;
```

3.1.3 Signal Destination Descriptor

The signal destination descriptor (*tSignalDestDescr*) describes into which Tx buffer a signal should be copied and to what position. The *position* member is six bits long, covering any bit position of the signal between 0 and 63. The *frame_no* member is ten bits long, allowing referencing up to 1024 Tx buffers. The signal destination descriptor occupies two bytes of memory space. Signal destination descriptor:

```
typedef const struct {
    unsigned int frame_no:10;
    unsigned int position:6;
} tSignalDestDescr;
```

3.1.4 Signal Descriptor

The signal descriptor (*tSignalDescr*) structure holds information about an incoming signal. It describes its position in the Rx buffer (*position*) and its size (*size*) in bits. The structure also contains information about the number of destinations the signal must be copied into upon reception (*dests_no*). Signal descriptor structure:

```
typedef const struct {
    unsigned int position:6;
    unsigned int size:6;
    unsigned int dests_no:4;
} tSignalDescr;
```

To describe the operations necessary to be performed after the reception of a signal, the signal descriptor is always followed in memory by the appropriate number of signal destination descriptors (Figure 12).

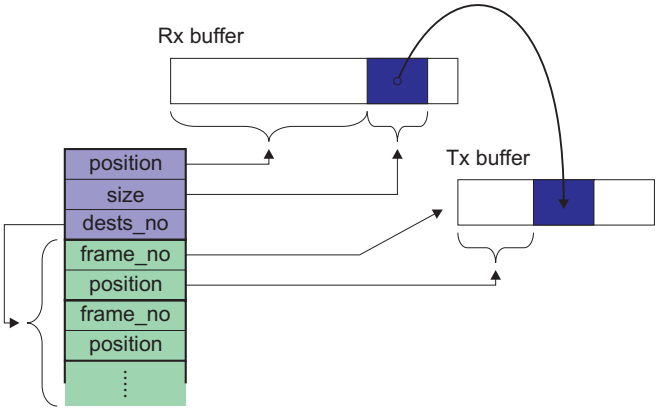

Figure 12. How Signal Destination Descriptors Follow Signal Descriptor

3.1.5 Rx Frame Descriptor

The Rx frame descriptor (*tRxFrmDescr*) describes an incoming frame. The structure contains:

- Frame ID (*ID*)
  - By which the incoming frame is associated with its descriptor.
- Expected data payload of the frame (*DataSize*)
  - An error handler executes when less than the expected number of bytes is received.
- Rx timer prescaler (*RxTimerPrescaler*) and Rx timer (*RxTimer*)
  - The prescaler is five bits long; however, a special algorithm allows decadic division ratios while keeping the XGATE load at a low level. The possible prescaler ratios are 1, 2, 10, and 20. (Section 3.2.5, “Handling of Rx and Tx Timers,” further explains the algorithm.)
- Number of signals contained in the frame (*SignalCount*) and a pointer to the first signal descriptor (*pSignalDescr*)
  - Figure 12 shows how signal destination descriptors follow the signal descriptor. If the Rx frame contains more than one signal, the second signal descriptor (and its destination descriptors) follows the first signal’s last destination descriptor. Figure 13 depicts the organization of the descriptors in memory.
- Timeout failure indicator enable (*RxToutFailEn*) and timeout failure indicator position (*RxToutFailPos*)
  - When an Rx timeout is detected, the gateway can set an indicator bit in the Tx buffer pointed to by the first signal destination descriptor of the first signal. The indicator is cleared when the Rx frame is received again. (See Section 2.3.1, “Reception,” and Section 3.2.5, “Handling of Rx and Tx Timers.”)
- Timeout counter (*RxToutCntr*) and timeout counter reload (*RxToutReload*)

  - The timeout counter is decremented every time the Rx timer expires (a timeout is detected). The timeout handler routine is called after the timeout counter is decremented down to zero.
- Byte copy indicator (*byte_copy*)
  - Enables faster copy of signals that span whole bytes. This feature is usually used for message gatewaying, where a single signal is defined which spans the whole data payload. This indicator speeds up the signal copy process. (See Section 4.1.1.2, “Signal Copy,” for performance implications.)

Rx frame descriptor structure:

```
typedef struct {
    unsigned int ID:11;                     /* Frame ID */
    unsigned int RxTimerPrescaler:5;        /* prescaller of the Rx Timer */
    tSignalDescr * pSignalDescr;            /* pointer to Signal descriptors */
    unsigned int DataSize:4;                /* expected data size for this frame */
    unsigned int SignalCount:8;             /* number of signals */
    tTimer RxTimer;                         /* Rx timeout */
    unsigned int byte_copy:1;               /* byte copy of data */
    unsigned int RxToutFailEn:1;            /* enable of the out-of-date bit assertion */
    unsigned int RxToutFailPos:6;           /* position of the out-of-date bit */
    unsigned int RxToutReload:4;            /* timeout counter reload */
    unsigned int RxToutCntr:4;              /* timeout counter */
} tRxFrmDescr;
```

The implementation of the two timeout conditions (Section 2.3.1, “Reception”) makes their timing depend on each other. The Rx timer load value defines the shorter timeout (after which the failure indicator is set). The longer timeout (after which the timeout handler routine is called) is defined as a multiple of the short timeout. The timeout counter reload defines how many short timeouts must occur before a long timeout is detected.

The Rx frame descriptors are stored in memory in an array. The search algorithm requires that the array is sorted (in ascending order):

1. By the number of the node through which the frame is received
2. By frame ID

A single node’s Rx frame descriptors are grouped. Within a group, the descriptors are sorted by frame ID. This enables the use of a fast search algorithm; the gateway functionality is not compromised.

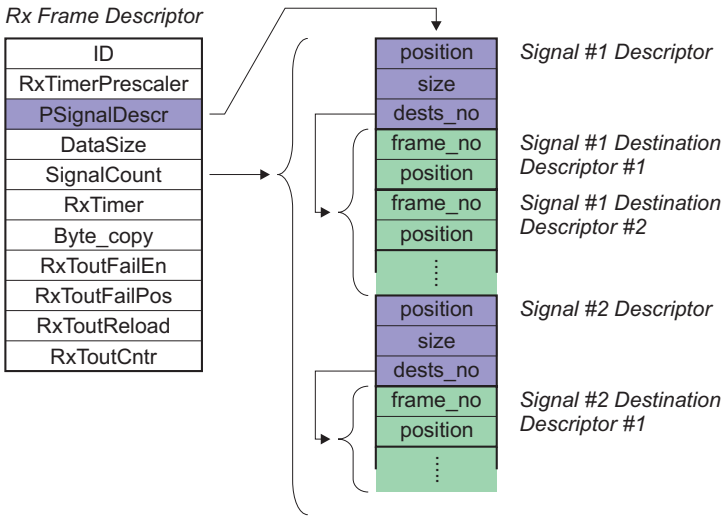

Figure 13. Memory Footprint of Signal and Signal Destination Descriptors

3.1.6 Tx Frame Descriptor

The Tx frame descriptor (*tTxFrmDescr*) describes the outgoing frames. The structure contains:

- Frame ID (*ID*) — Transmitted as part of frame.
- Node number (*node*) — Identifies which CAN or LIN node the frame should be transmitted through.
- Tx timer prescaler (*TxTimerPrescaler*) and Tx timer (*TxTimer*) — Specify transmission period in periodic transmission modes (Section 2.3.2, “Transmission” and Section 3.2.5, “Handling of Rx and Tx Timers”).
- Data size (*DataSize*) — Specifies the number of bytes to be transmitted as the data payload of the frame.
- Immediate transmission upon any signal reception indicator (*TxonRx*). (Section 2.3.2, “Transmission”)
- Immediate transmission upon any signal value change indicator (*TxonDataChg*). (Section 2.3.2, “Transmission”)
- Transmission scheduled indicator (*TxScheduled*) — Flag serving as an interlock to ensure the frame can be scheduled for transmission only once before it is transmitted through the appropriate node. (Section 3.2.4, “Transmission of Frames”).

Tx frame descriptor structure:

```
typedef struct {
    unsigned int ID:11;                  /* ID of the frame */
    unsigned int node:5;                 /* node number */
    tTimer TxTimer;                      /* Tx periodic timer */
    unsigned char * Data;                /* pointer to the data buffer */
    unsigned int TxTimerPrescaler:5;     /* prescaller of the Rx Timer */
    unsigned int DataSize:4;             /* data size for this frame */
    unsigned int TxonRx:1;               /* transmit frame on data reception */
    unsigned int TxonDataChg:1;          /* transmit frame on data change */
    unsigned int TxScheduled:1;          /* transmission is scheduled */
} tTxFrmDescr;
```

3.1.7 Node Descriptor

The node descriptor (*tNodeDescr*) describes the individual nodes through which the frames are received and transmitted. The structure contains:

- Address of the peripheral (*periph_addr*)
  - The value is the MSCAN peripheral (CAN node) address or the LIN node descriptor (LIN node) address.
- Pointer to the first Rx frame descriptor (*rx_idx_start_p*) and the number of frame descriptors (*rx_idx_cnt*) belonging to this node
  - These pieces of information delimit the space within the Rx descriptor table the search algorithm needs to move through to identify frame descriptor of any received frame.
- Peripheral type (*periph_type*)
  - Identifies peripheral type the node uses.
- Transmission queue pointer (*TxBuffer*), index of the first item (*TxBufferTake*), index of the first empty slot (*TxBufferAdd*), and size of the queue (*TxBufferSize*)
  - Manage transmission queue. See Section 3.2.1, “Reception of Frames” and Section 3.2.4, “Transmission of Frames,” for further details.

Node descriptor structure:

```
typedef struct {
    unsigned int periph_addr;            /* address of the peripheral */
    tRxFrmDescr* rx_idx_start_p;         /* pointer to the first Rx descriptor */
    unsigned int rx_idx_cnt;             /* number of Rx descriptors for this node */
    unsigned char periph_type;           /* peripheral type */
    unsigned char TxBufferAdd;           /* index after the last entry in the queue */
    unsigned char TxBufferTake;          /* index to the first entry in the queue */
    unsigned char TxBufferSize;          /* size of the Tx queue */
    tTxFrmDescr** TxBuffer;              /* pointer to the Tx queue */
} tNodeDescr;
```

Implemented peripheral (node) types:

```
enum periph_types {
    MsCan,
    Lin
};
```

3.1.8 LIN Data Structures

Full description of LIN master node implementation on the HCS12X architecture is beyond the scope of this document. The gateway application uses LIN master implementation described in Freescale's application note, AN2732: "Using XGATE to Implement LIN Communication on HCS12X." To calculate the required memory size of the LIN node descriptor structure, see LIN node descriptor:

```
typedef struct {
    tSCI* pSCI;                         /* pointer to the SCI peripheral */
    unsigned int checksum;              /* checksum */
    unsigned char data[8];              /* buffer for Tx and Rx data */
    unsigned char Id;                   /* LIN identifier to be transmitted */
    unsigned char dir:1;                /* direction: Rx = 0; Tx = 1 */
    unsigned char len:4;                /* length of frame */
    unsigned char state;                /* state of the LIN state machine */
    signed char timer;                  /* Rx timeout counter */
    unsigned char* data_p;              /* pointer to LIN frame data */
} tLINnode;
```

3.2 Algorithms

This section describes the algorithms used in the gateway application. The gateway application has five entry points:

- MSCAN Rx interrupt — MSCAN module raises this interrupt request when it receives a message. The algorithm processing incoming frames is discussed in Section 3.2.1, "Reception of Frames."
- MSCAN Tx buffer empty interrupt — MSCAN module raises this interrupt request when a Tx buffer becomes empty after a successful frame transmission. The algorithm used for transmitting frames is discussed in Section 3.2.4, "Transmission of Frames."
- Gateway timing interrupt — The gateway application uses one channel of the periodic interrupt timer (PIT) to time all Rx and Tx operations. The algorithm for managing the Rx and Tx timers is described in Section 3.2.5, "Handling of Rx and Tx Timers."
- SCI Rx buffer full / Tx buffer empty interrupt — The LIN master algorithm uses SCI interrupts. The operation of the LIN algorithm is described in Freescale's application note, AN2732: "Using XGATE to Implement LIN Communication on HCS12X."
- LIN timing interrupt — The LIN algorithm uses another PIT channel to detect occurrences of LIN Rx timeouts. The operation of the LIN algorithm is described in Freescale's application note, AN2732: "Using XGATE to Implement LIN Communication on HCS12X."

3.2.1 Reception of Frames

When we consider the data flow, then the reception algorithms are the entry points of the gateway application. There are two reception algorithms, one for each communication protocol: CAN and LIN.

3.2.2 Reception of CAN Frames

Figure 14 outlines the algorithm for processing received CAN frames.

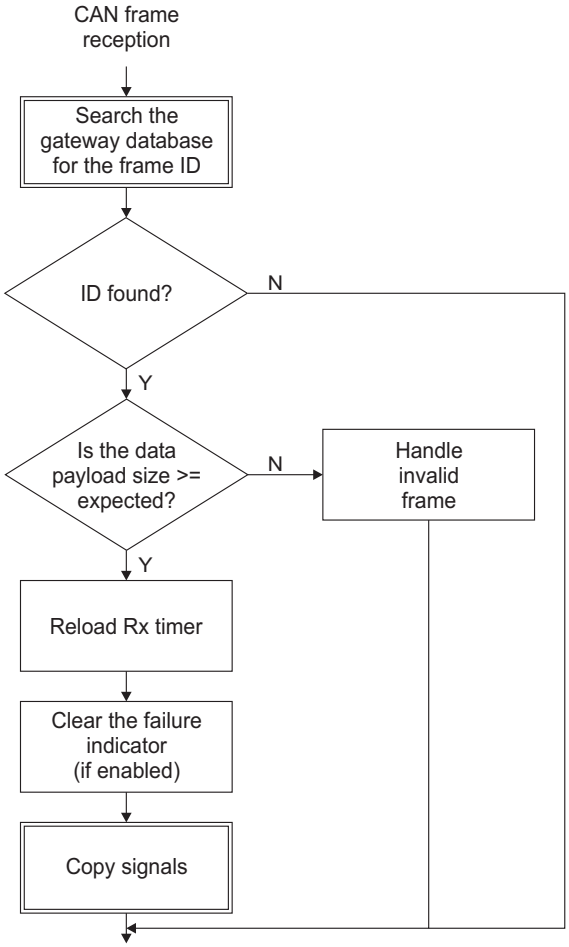

Figure 14. Algorithm for Processing of Received CAN Frames

On the CAN bus, frames are received asynchronously. The gateway application cannot rely on specific frames being received at specific times or in a specific order. The only identification is the received frame's ID. Therefore, the algorithm first searches the gateway database of Rx frame descriptors to find information about the received frame (Section 3.2.2.1, "ID Search"). If no information about the frame is found, the frame is not further processed, and the algorithm finishes execution.

If an Rx frame descriptor with a matching ID is found, the algorithm checks the data payload of the received frame. If the frame contains less than the expected number of bytes, the invalid frame handler routine (*RxInvalidHandler*) is called, and the frame is not further processed. The action taken in case of invalid frame reception may differ depending on the information stored in the frame (ignore the frame, request re-send, report an error, etc.). The invalid frame handler contents should be tailored to suit the gateway application requirements.

After the frame passes the payload size check, the gateway reloads its Rx timer (that is, the reception timeout starts running again). If a failure indicator is enabled for the frame, the algorithm clears it.

The final step in the algorithm is the gateway's main task. The data payload is broken into individual signals and copied into their destinations. More details about the signal copy algorithm can be found in Section 3.2.2.2, "Signal Copy."

3.2.2.1 ID Search

The algorithm that performs the search for an Rx frame descriptor with an ID matching the currently received frame (*RxFindFrmId*) is a variation on the binary search algorithm. The binary search algorithm relies on the array elements being sorted. This is why the Rx frame descriptors must be sorted by their frame IDs (in ascending order). *RxFindFrmId* function:

```
tRxFrmDescr *RxFindFrmId(int FrmId, tNodeDescr *node) {
    tRxFrmDescr *base=(node->rx_idx_start_p);
    int i;
    int result;
    tRxFrmDescr *p;
    for (i = node->rx_idx_cnt; i != 0; i >>= 1) {
        p = base + (i >> 1);
        result = FrmId-(p->ID);
        if (result==0) return(p);           /* if spot on, return the pointer */
        if (result>0) {                     /* moving right? */
            base=p+1;                       /* adjust the index and base */
            i--;
        }
    }
    return (NULL);
}
```

Its parameters are a pointer to a node descriptor of the node through which the frame was received and the received frame's ID. It returns a pointer to the matching Rx frame descriptor or a NULL pointer in case an Rx frame descriptor with a matching ID is not found. The function does not search the whole Rx frame descriptor table; it searches the portion of the table containing descriptors for the specified node.

3.2.2.2 Signal Copy

The algorithm for the signal copy routine (Figure 15) consists of two nested loops. The outer loop cycles through all the signals present in the source data. For every signal, the inner loop cycles through all destinations for the frame. For each signal destination, the signal is extracted from the source data buffer and copied to the destination Tx buffer.

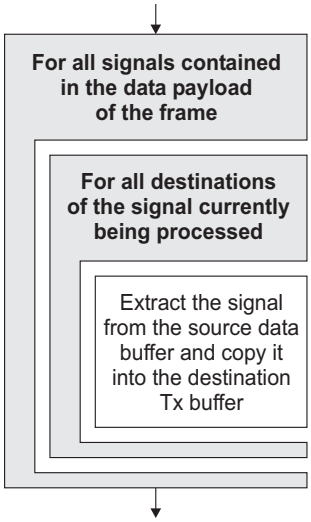

Figure 15. Algorithm for Copying Signals

3.2.3 Reception of LIN Frames

Unlike CAN, the LIN bus is based on a single master concept. The master in the system initiates all transfers on the bus. The gateway application acts as master on the attached LIN buses and is in full control of the operations.

Because the gateway application initiates reception of frames on the LIN buses, searching the gateway database for a corresponding Rx frame descriptor is unnecessary, as the application knows which frame is being received. The architecture of the LIN drivers also ensures that only correctly received Rx frames containing the expected amount of data are submitted to the gateway application for further processing.

The algorithm flow (Figure 14) is simplified for the LIN buses because ID-search and the payload-size checking can be omitted. The remaining steps are the same.

3.2.4 Transmission of Frames

The frame transmission algorithm (*FrameTransmit*) does not physically transmit the frame in question. It operates on the transmission queues and inserts a pointer to the frame descriptor into the appropriate queue. Figure 16 outlines the algorithm.

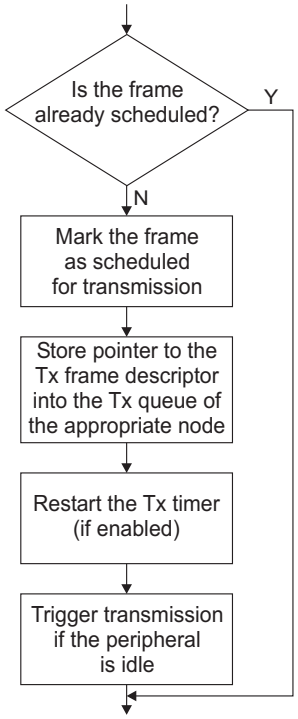

Figure 16. Frame Transmission Algorithm

When the routine executes, it marks the frame as scheduled for transmission. The *TxScheduled* flag serves as an interlock to ensure the frame cannot be inserted into the transmission queue more than once. If any signal contained in this frame is updated before the frame moves to the front of the queue and is transmitted, there is no need to insert the frame into the queue again. When transmitted, it will contain the newest available value of the signal.

After the frame is marked as scheduled, a pointer to its Tx frame descriptor is inserted into the transmission queue buffer of the appropriate node. The *TxBufferAdd* index is then incremented to point to the first unused element in the buffer. To prevent overflows, the queue buffer must be long enough for the worst case scenario. In this particular implementation, the buffer is sized to have enough space for all the frames of the particular node. It is possible to schedule all frames at once without an overflow occurring in the queue buffer.

The algorithm then reloads the Tx timer to start a new period in case of periodic transmissions. The Tx timer is reloaded when the frame is scheduled for transmission and not when it is actually transmitted. This means that traffic on the bus—or a long queue delaying the actual transmission—does not interfere with the timing of the periodic transmission process, and a stable period is maintained.

If the peripheral in question is idle, the last step in the transmission algorithm is triggering the actual transmission. When there are no more frames to transmit (the queue is empty), the interrupt service routine that loads the peripheral with frames according to the transmission queue disables the transmit buffer empty interrupt. The transmission algorithm must recognize this situation and enable the interrupt source to allow the interrupt routine to execute and load the peripheral with the newly scheduled frame.

3.2.5 Handling of Rx and Tx Timers

A single routine (*GatewayTick*) manages all the timers within the gateway application. As the name of the routine suggests, it executes periodically in response to an interrupt signal from an on-chip hardware timer. Whether the underlying hardware timer generates timebase with a period of 1 ms, 10 ms, or even longer is irrelevant—the algorithms work exactly the same. To minimize the XGATE load, set the timebase to as long a period as possible (typically 10 ms).

Each timer has its own prescaler. Typical gateway requirements set the periodic transmission frequencies and reception timeouts to values easily understood by humans (for example, 500 ms rather than 512 ms). To support human-readable numbers, the prescaler allows the timer to be updated on every 1st, every 2nd, every 10th or every 20th opportunity. A single static *counter* variable implements the prescaler functionality. The *counter* variable updates at the end of the interrupt routine (after all timer algorithms have executed). The counter is an eight-bits long variable divided into two, 4-bit nibbles. On every update, the *counter* value is incremented. If the lower nibble equals 10, the nibble is cleared and the upper nibble is incremented. This simplifies implementation of the decimal prescalers of the individual timers to a masking (bitwise AND) operation and shortens the required execution time.

Each algorithm also checks whether the timer value equals zero. A value of zero is not valid during normal timer operation and is considered to have a special meaning (timer disabled). This makes it easy for the software layers above the gateway application to start and stop timers, as required.

If the timer value is non-zero, the timer is decremented. If the timer equals zero after it is decremented, it is considered expired. After expiration, the timer is reloaded and an action is taken.

The action depends on the individual timer. Timers within the gateway application have three uses:

- Tx timer
- CAN Rx timer
- LIN Rx timer

3.2.5.1 Tx Timers

Tx timers are used for periodic transmission of frames. Figure 17 shows the algorithm for processing Tx timers.

All the Tx timers are processed sequentially in a loop. When a Tx timer expires, it is reloaded and the frame the timer belongs to is scheduled for transmission.

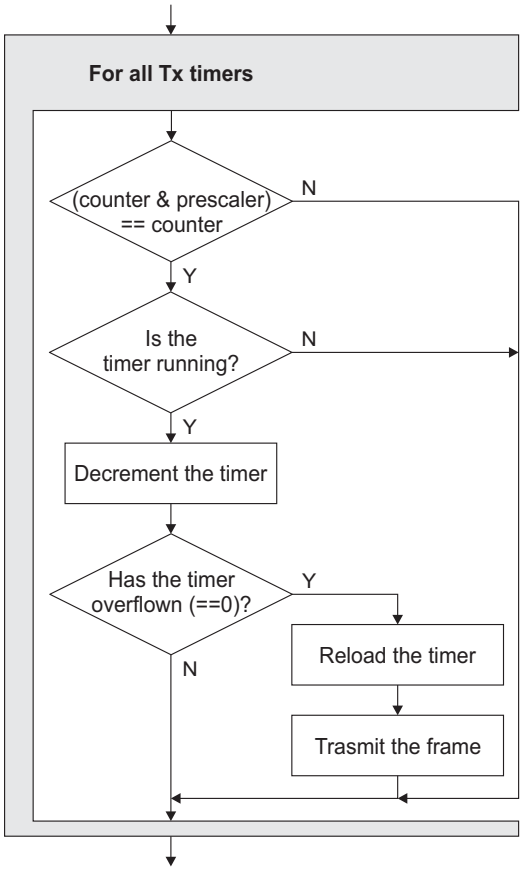

Figure 17. Tx Timer Processing

3.2.5.2 CAN Rx timers

The Rx timers are used for detecting timeouts in CAN frame receptions. The algorithm for processing Rx timers is slightly more complicated than for Tx times because it needs to handle the two timeout actions described in Section 2.3.1, “Reception.” Figure 18 shows the algorithm. Every time the CAN Rx timer expires, the out-of-date indicator is set (if enabled), and the appropriate timeout counter is decremented. When the timeout counter reaches zero, the timeout processing routine is called.

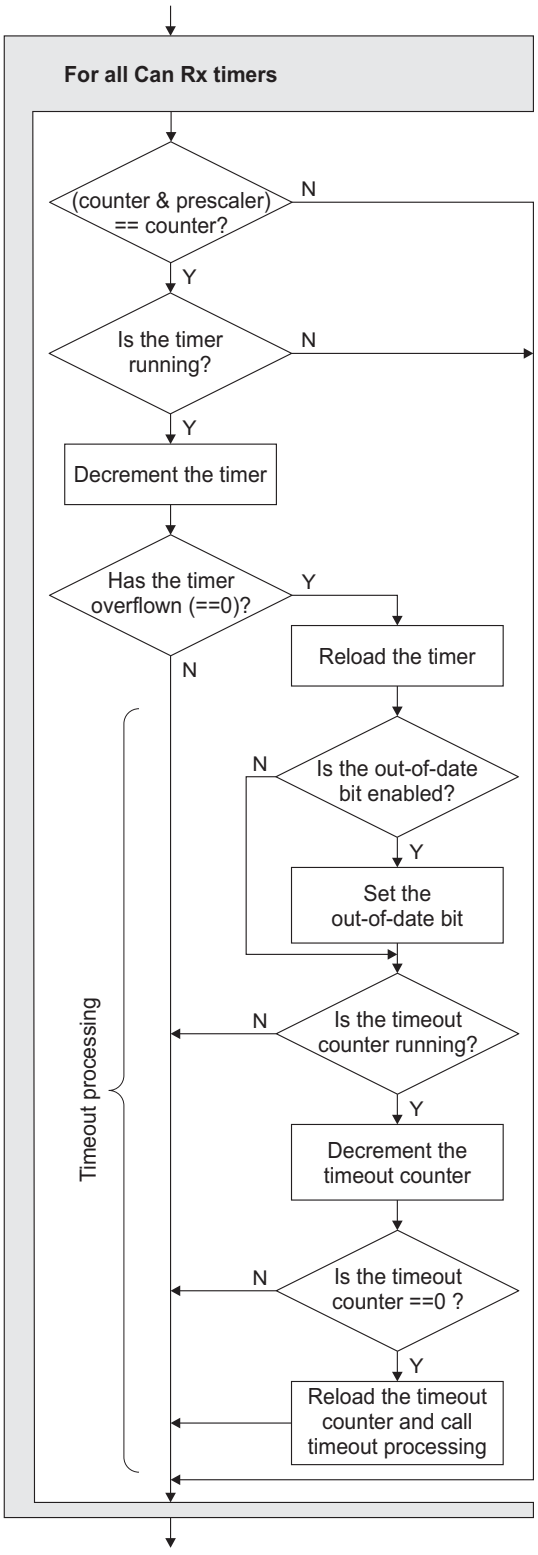

Figure 18. CAN Rx Timer Processing

3.2.5.3 LIN Rx Timers

Because of the LIN protocol (Section 2.3.3, “LIN Considerations”), LIN Rx timers are similar to Tx timers. The only difference between the algorithms is in the fact that the Rx frame descriptor does not directly indicate the node it belongs to; a quick search is necessary to identify the node through which the frame will be received. The search does not represent a major load for the XGATE: the number of node descriptors it needs to visit is low (limited by the number of LIN nodes the application uses). Figure 19 shows the algorithm for processing the LIN Rx timers.

It is not necessary to detect any timeouts associated with the LIN frames at the gateway level; the timeouts are detected within the LIN driver itself (see [2]). When the LIN driver detects a timeout during frame reception, it executes the same timeout algorithm that is in place for CAN frames (see bottom half of Figure 18).

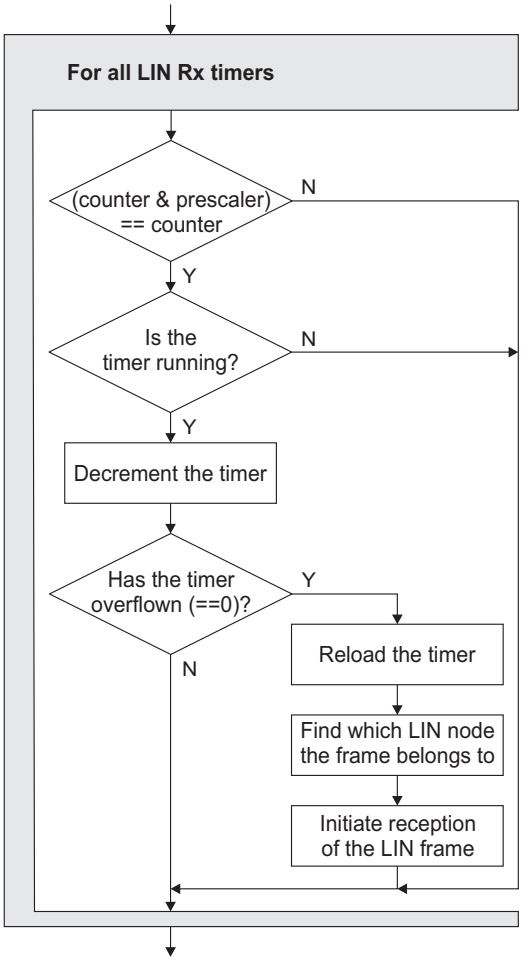

Figure 19. LIN Rx Timer Processing

3.3 Limitations of This Gateway Implementation

The sample gateway application described in this document has limitations (Table 1) caused by choices made during the creation of the example; however, requirements exceeding these limitations can be accommodated if an appropriate change is made to the source code. Such changes would mainly impact the memory footprint of the application; the XGATE load created by the application would vary only slightly and could increase or decrease based on how the change is implemented (for example, increasing the maximum frame ID length from 11 bits to 16 bits increases the required memory size but simplifies processing because no bit manipulations would be needed to access the ID).

Table 1. Limitations

Parameter	Allowed range
Frame data size	0 – 15 bytes
Signal size	1 – 64 bits
Signal position	0 – 63
Number of Tx buffers	0 – 1024
Rx/Tx timer period	1 – 5120 ticks (51.2s max. @ 10 ms time base)
Frame ID length	1 – 11 bits

4 Performance Analysis

The CodeWarrior™ compiler/debugger tools version 4.5 were used to measure the gateway application's performance and required memory size. The example project used for the measurement is available in a zip file associated with this application note. Execution times of the different parts of the gateway application were measured while the HCS12X CPU performed infrequent accesses into the on-chip RAM.

4.1 XGATE Load

The XGATE performance required by the gateway application can be analyzed by studying the requirements of the individual threads XGATE executes. As outlined in Section 3.2, "Algorithms," the gateway application has five separate entry points (five different threads to be analyzed). The load created by the threads handling these entry points are discussed in this section.

4.1.1 CAN Rx Thread

The CAN Rx thread executes after the reception of each CAN message.

4.1.1.1 ID Search

The first step in the CAN Rx algorithm is to find the Rx frame descriptor with a matching ID (see Section 3.2.2, "Reception of CAN Frames," and Section 3.2.2.1, "ID Search"). The search time depends on the portion size of the Rx descriptor table to be searched and also on the descriptor position with a matching ID within the table. Figure 20 shows the worst case scenarios (expressed in bus cycles) for search sizes up to 200.

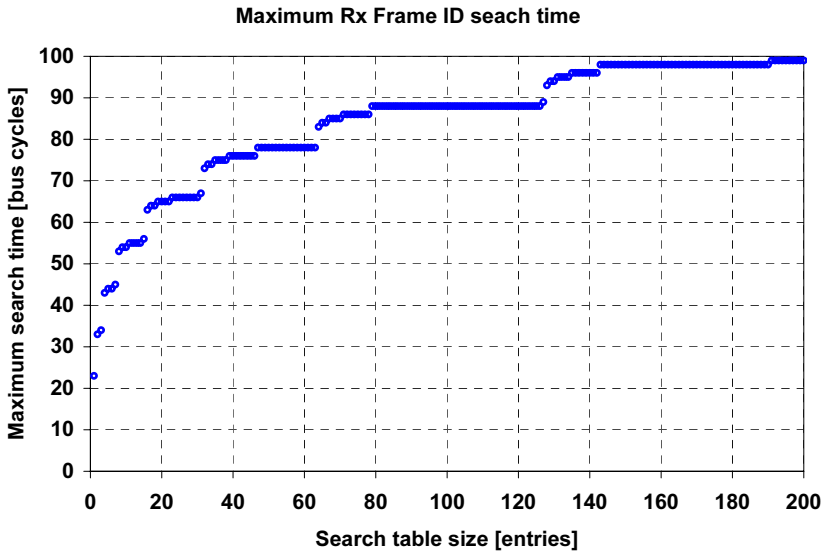

Figure 20. Maximum ID Search Time for Different Sizes of Rx Descriptor Table

The maximum search time trend closely matches the number of iterations required by the underlying binary search algorithm (the number of required iterations is at most $\log_2 N$, where N is the number of elements of the array).

Only the Rx descriptors belonging to the particular CAN node are searched (as opposed to searching the whole Rx descriptor table containing descriptors for all CAN nodes).

4.1.1.2 Signal Copy

After the appropriate Rx frame descriptor with matching ID is found, and the size of the received frame is verified, the signals are copied into their destination buffers. This process is the most intensive task the gateway application performs and represents the majority of required processing time.

There are two options governing the actual copying of the signals. Each Rx frame descriptor indicates whether the particular frame data payload should be copied with bit or byte accuracy (*byte_copy*). An additional compile-time option for signals copied with bit accuracy governs whether the data payload is in big endian or little endian format.

Figure 21 shows the execution time of the byte accurate signal copy function (*CopyDataChk*).

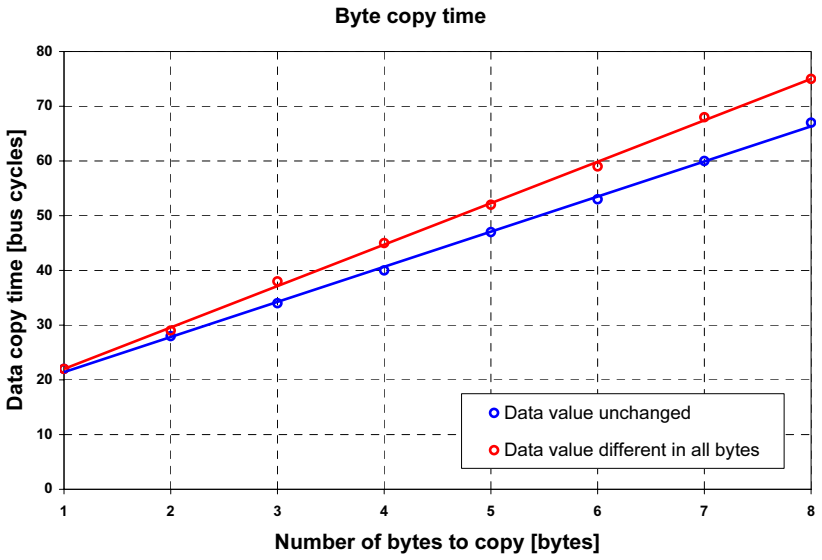

Figure 21. Byte Copy Time for Different Data Sizes

Copying data with bit accuracy is more demanding in terms of the number of required bus cycles. There is little difference between little and big endian data formats in terms of required performance. Figure 22 shows the execution time of the big endian bit accurate signal copy function (*CopySignalChkBE*).

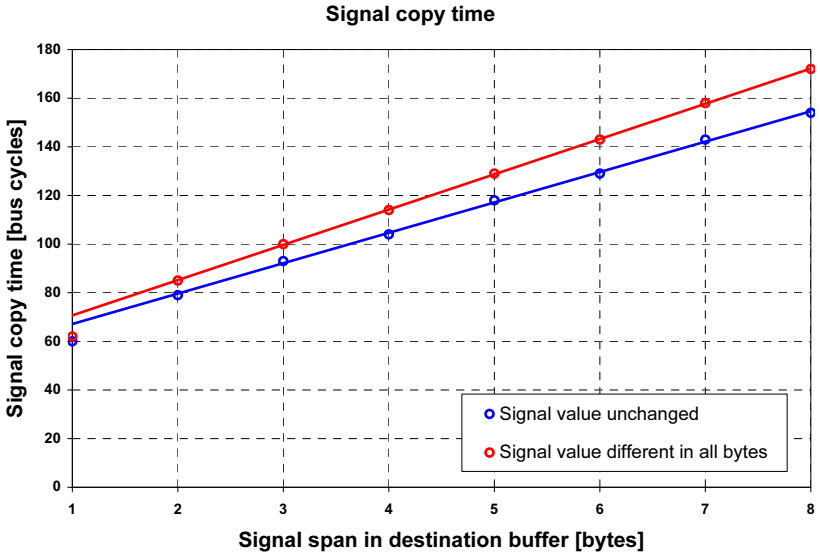

Figure 22. Bit Accurate Signal Copy Time for Different Data Sizes

It is more demanding to copy a long signal than a short signal. The parameter, which dictates the execution time in this implementation of the functions, is the signal span in the destination data buffer. By span, we mean the number of bytes the signal occupies. Figure 23 shows examples of signals spanning different numbers of bytes.

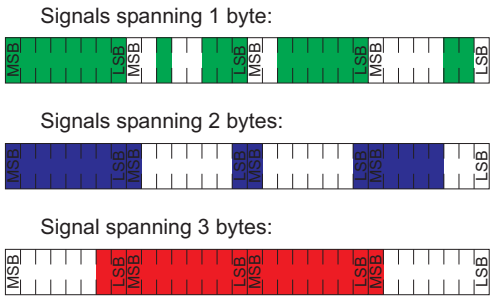

Figure 23. Signals Spanning 1, 2 and 3 Bytes

The algorithm of the bit accurate signal copy function consists of a loop that executes once for every byte of signal span in the destination buffer. This is why the execution time grows linearly with the signal span. The function is optimized for processing short signals that fit into a single byte. The execution time for signals spanning one byte is slightly shorter than the linear trend predicts.

As discussed in Section 2.3.2, “Transmission,” the gateway application can be configured to transmit frames when values of their signals change. Changes in signal values can be detected only during the copy process; therefore, execution time varies depending on the signal value. Figure 21 and Figure 22 show the best and worst case scenarios (copy of an unchanged signal and a signal that differs in all destination bytes). In general, the function’s execution time is between the two depicted extremes.

As outlined in Section 3.2.2.2, “Signal Copy,” the signal copy algorithm itself (*CopySignals*) consists of two nested loops. The outer loop runs through all the signal descriptors for the frame while the inner loop runs through all destinations of the signal currently processed. Figure 24 shows an example of how the execution time of this algorithm depends on the number of signals and their destinations.

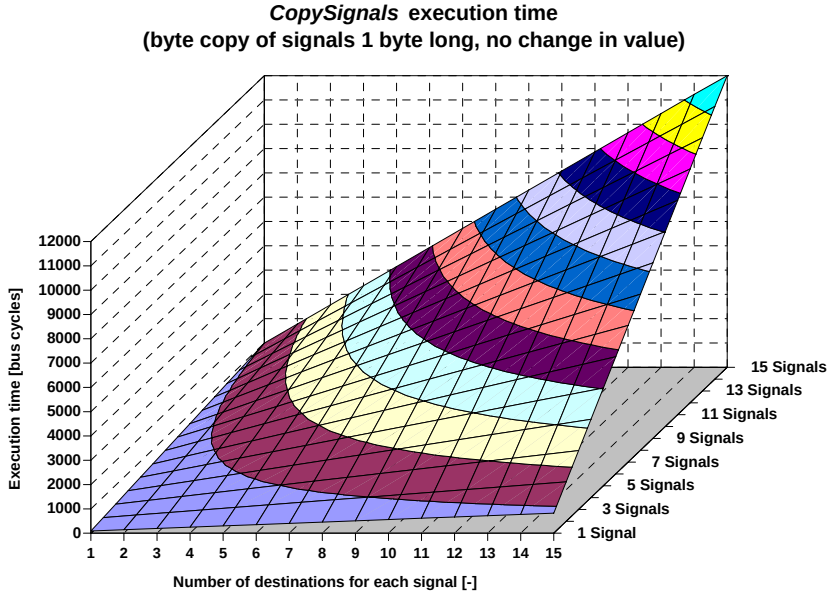

Figure 24. Execution Time of *CopySignals*

As shown in Figure 24, the execution time grows linearly with the number of signals and the number of signal destinations the algorithm must process. The execution time can, therefore, be approximated by the formula found in Equation 1. The parameters in this equation are the number of signals in the Rx frame and the total number of destinations for all the signals combined.

$$T_{CopySignals} = 26 + 12 \cdot N_{signals} + 32 \cdot N_{destinations} \quad [\text{bus cycles}] \qquad \textit{Eqn. 1}$$

The execution times of *CopyDataChk* and *CopySignalChkBE* are excluded from the equation and must be added to obtain the total time the signal copy process requires. Combining Equation 1 with the copy times (Figure 21 and Figure 22) enables the software designer to estimate the execution time the signal copy algorithm requires.

4.1.1.3 Frame Transmission Scheduling

Procedure *FrameTransmit* schedules frames for transmission (see Section 3.2.4, “Transmission of Frames”). The execution time of the procedure depends on whether the particular frame is already scheduled for transmission. Table 2 shows the procedure’s execution times. Execution time of the LIN frame setup procedure (*LinFrameSetup*)—in case the scheduled frame is a LIN frame—is excluded from the values in Table 2.

Table 2. Execution Times of *FrameTransmit* Procedure

Condition	Execution time [Bus Cycles]
Frame already scheduled	12
Frame not scheduled yet	40

The *LinFrameSetup* procedure executes an algorithm that starts operation of the LIN peripheral if it was idle when the frame is scheduled for transmission. Its execution time depends on whether an Rx or a Tx LIN frame is being processed. Table 3 shows the procedure’s execution times.

Table 3. Execution Times of *LinFrameSetup* Procedure

Condition	Execution Time [Bus Cycles]
Tx frame at top of queue	70
Rx frame at top of queue	42
Transmission queue empty	11

4.1.2 CAN Tx Thread

The Tx thread (*MsCanTxEmptyIsr*) executes in response to the MSCAN Tx buffer empty interrupt request. Execution time of this interrupt service routine depends on whether any frames are scheduled for transmission or whether the transmission queue is empty. When the queue is empty, the interrupt routine only disables the interrupt source because there are no more frames to transmit. Table 4 shows execution times.

Table 4. Execution Times of *MsCanTxEmptyIsr* Interrupt Service Routine

Condition	Max. execution time [bus cycles]
Frame(s) scheduled for transmission	53
Transmission queue empty	12

4.1.3 Rx and Tx Timer Handling Thread

As described in Section 3.2.5, "Handling of Rx and Tx Timers," the timers are used for three different purposes within the gateway application. These groups of timers are processed independently; you can separately analyze the execution time of the timer handling thread (*GatewayTick*) for the three groups. The overall execution time is an addition of the processing times required for the three timer groups.

The execution times detailed in the following sections were measured under the following conditions:

- All timers have reload value equal to 202.
- The expiration of individual timers is staggered (only one timer expires during one execution of the thread).
- Calls of other functions within the gateway application (such as transmission scheduling) were excluded during the measurements.

4.1.3.1 Tx Timers

Figure 25 shows the processing times required for different numbers of Tx timers with different prescalers.

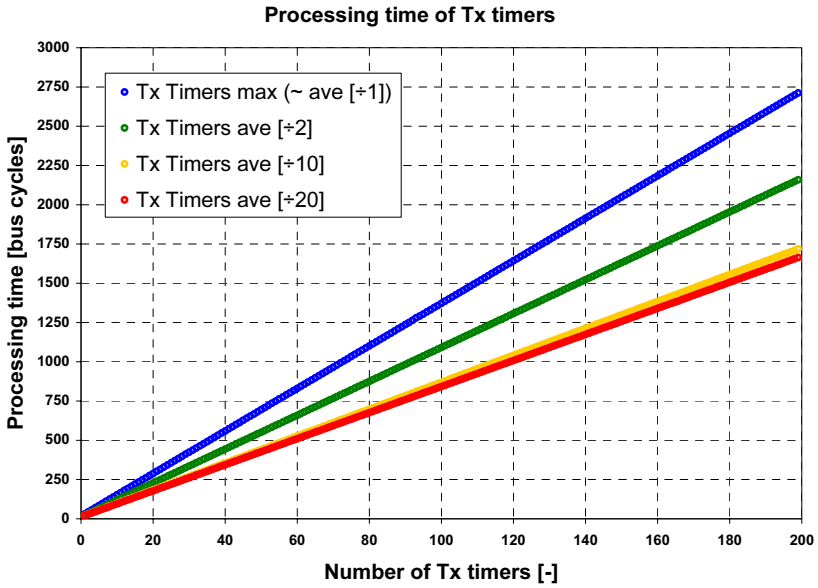

Figure 25. Processing Time Required for Tx Timers

The maximum execution time is independent of the prescaler selection and depends on the number of timers being processed. The average execution time for prescaler values greater than 1 is lower because the timers are not decremented in every iteration of the algorithm.

4.1.3.2 CAN Rx Timers

In terms of processing complexity, the CAN Rx timers are almost identical to the Tx timers. Figure 26 shows the processing times required for different numbers of CAN Rx timers with different prescalers.

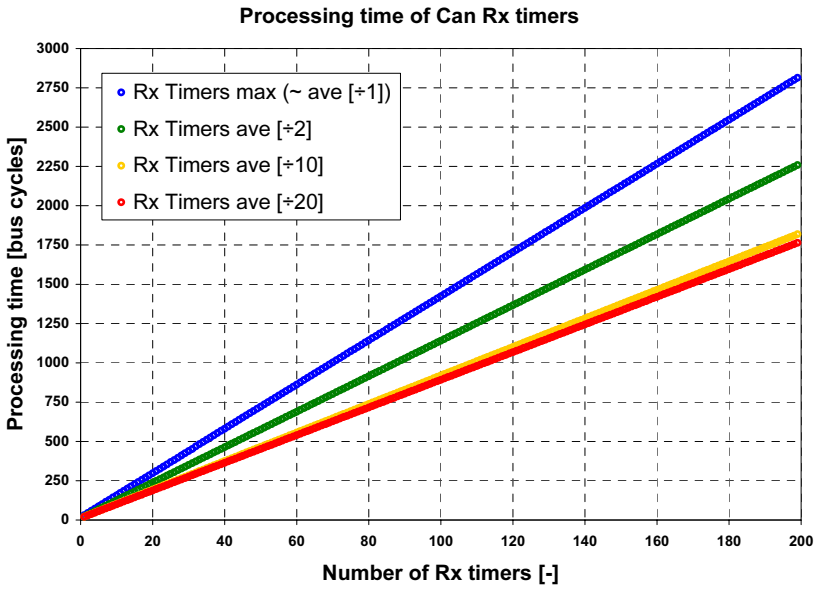

Figure 26. Processing Time Required for Can Rx Timers

4.1.3.3 LIN Rx Timers

Processing of LIN Rx timers includes a search to identify which LIN node the timer belongs to. The search time is short, but because it is only performed when a timer expires, the average processing time for prescaler value of 1 is no longer equal to the maximum processing time.

The search time is the longest when the expired timer belongs to the last LIN node. For the purpose of this measurement, six LIN nodes were defined, and all the timers were assigned to the last LIN node (this situation represents the worst case scenario for gateway application with six LIN interfaces).

Figure 27 shows the processing times required for different numbers of LIN Rx timers with different prescalers.

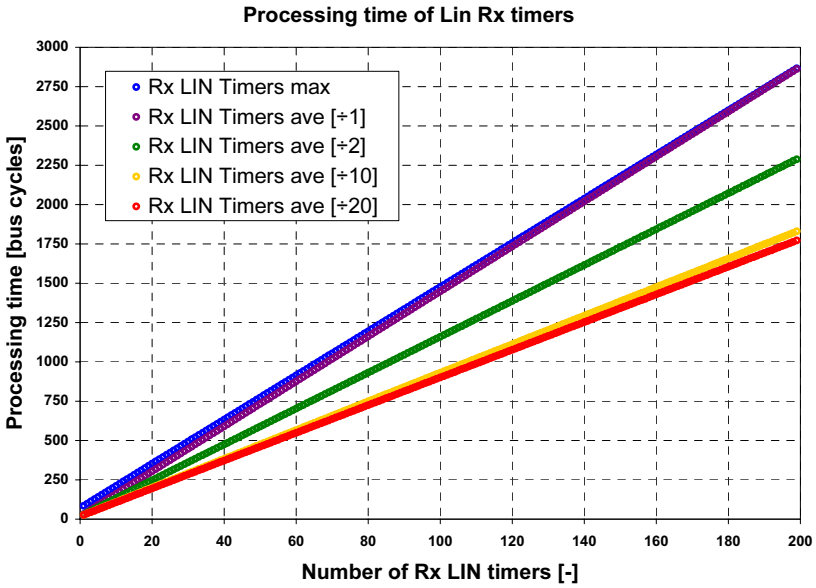

Figure 27. Processing Time Required for Lin Rx Timers

4.1.4 LIN Master Algorithm

The LIN master algorithm is implemented as a simple state-machine which responds to SCI Tx buffer empty and Rx buffer full interrupts (*LinSciIsr*). Execution time of the algorithm depends on the state of the algorithm. Table 5 shows the execution times in different states.

Table 5. Execution Time of The Lin Master Algorithm

State	Execution time [bus cycles]
Send break & sync characters	21
Send ID (when transmitting)	22
Send ID (when receiving)	25
Receive sync (when receiving)	19
Receive ID (when receiving)	24
Transmit data	36
Receive data	37
Transmit checksum	28
Receive checksum and set up new frame (queue empty)	58
Set up new frame (when transmitting) (Tx frame at top of queue)	89
Set up new frame (when transmitting) (Rx frame at top of queue)	63
Set up new frame (when transmitting) (queue empty)	32

The LIN master algorithm calls the *LinFrameSetup* procedure to set up transmission or reception of a new frame after processing of the current frame is complete (see Section 4.1.1.3, “Frame Transmission Scheduling”).

4.1.5 LIN Timeout Detection

The LIN timeout detection algorithm (*LinTimeoutTick*) executes at a frequency equal to 1/5 of the symbol rate of the LIN peripherals (all LIN peripherals are expected to run at the same speed, typically 19,200 or 9600 baud).

The algorithm execution time depends on two parameters: the number of LIN nodes used by the gateway application and the number of LIN nodes currently receiving frames from slave devices (and, therefore, in need of timeout detection). Figure 28 shows the algorithm execution times. The execution times in the graph are based on cases where the timeout detection is active for the selected number of LIN nodes, but no timeout has been detected (all data from slave devices are received on time).

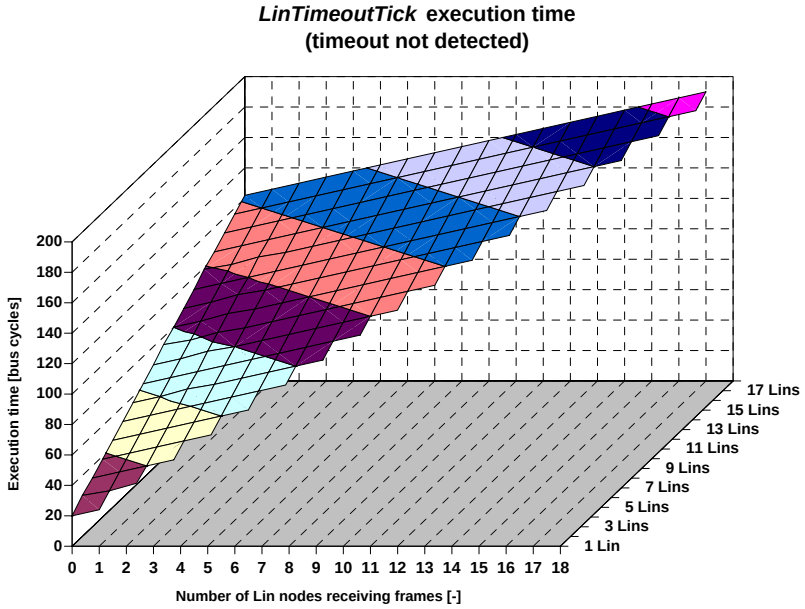

Figure 28. Execution Time of *LinTimeoutTick*

Figure 28 shows how the execution time grows linearly with the number of LIN nodes and receiving LIN nodes. Equation 2 shows how to approximate the execution time. The parameters in this equation are the number of LIN nodes the gateway application uses and the number of LIN nodes receiving frames from slave devices.

$$T_{LinTimeoutTick} = 15 + 6 \cdot N_{LinNodes} + 4 \cdot N_{RxLinNodes} \quad [\text{bus cycles}] \qquad \textit{Eqn. 2}$$

The algorithm calls the *RxTimeoutHandler* routine when it detects a timeout on any receiving LIN node. In such a case, the execution time is longer than indicated above.

4.2 Required Memory Size

The CodeWarrior compiler version 4.5 was used to determine the size of the individual components of the gateway example application.

4.2.1 Code Size

The gateway database defines the behavior of the fundamental algorithms of the gateway application. The size of the code is constant and different requirements are accommodated by changing the gateway database. Only a few higher-level functions must be fine-tuned based on the actual requirements. Such functions are marked in the tables below.

Table 6 details the size of code for the CPU. All the CPU functions perform initialization activities and are only executed once after start-up.

Table 6. Code Size for the CPU

Function / Procedure	Size [bytes]
GatewayInit	78
InitPit	49
InitSci	29
InitMsCan	83
SetIntPrio	22
TOTAL	**261**

Table 7 shows gateway code size for the XGATE. The XGATE also executes the LIN driver. Table 8 shows code size of the XGATE LIN driver.

Table 7. Code Size for the XGATE

Function / Procedure	Size [bytes]
RxFindFrmId	58
CopySignalChkBE	264
CopyDataChk	42
CopySignals	212
BitAssign	48
RxTimeoutHandler	placeholder for user code
RxHandler	placeholder for user code
RxInvalidHandler	placeholder for user code
GatewayRxTimeout	94
GatewayTick	278

Table 7. Code Size for the XGATE (continued)

Function / Procedure	Size [bytes]
FrameTransmit	108
MsCanTxEmptyIsr	110
MsCanRxFullIsr	136
TOTAL	**1350**

Table 8. Code Size of the XGATE LIN driver

Function / Procedure	Size [bytes]
LinFrameSetup	210
LinSciIsr	450
LinTimeoutTick	96
TOTAL	**756**

4.2.2 Data Size

In a reasonably sized gateway application, the data structures are larger than the application code. The gateway database size depends on the signal routing requirements. It is only possible to document the sizes of the individual database components (descriptors), and you must calculate the required memory size based on the numbers of the descriptors required to satisfy the particular requirements. Table 9 shows the descriptor sizes.

Table 9. Sizes of Gateway Descriptors

Descriptor / variable	Use	Size [bytes]
tNodeDescr	Member of NodeDescrs array; describes a node (LIN or MSCAN)	12
tLINnode	Member of LinNodeDescrs array; describes additional properties of LIN nodes	18
tTxFrmDescr	Member of TxTable array; describes a Tx frame	8
tRxFrmDescr	Member of RxTable array; describes an Rx frame	10
tSignalDescr	Describes a signal (size and position in the Rx frame)	2
tSignalDestDescr	Describes a signal destination (Tx frame number and position)	2
Frm??Data	Data buffers (one for each Tx frame)	size of each buffer depends on the payload size of the particular Tx frame
Node?TxBuffer	Transmission queue buffers (one for each node)	see text below

The transmission queues must be long enough to avoid overflow even when all the frames of a node are scheduled for transmission. The queue must be long enough to hold pointers to all frames that can be scheduled plus one. On MSCAN nodes, only Tx frames can be scheduled for transmission; however, on LIN nodes, both Tx and Rx frames go through the scheduling process. On the XGATE co-processor, all pointers are 16 bits wide.

4.2.2.1 Data Size Calculation — an Example

This example demonstrates how data size can be calculated based on the gateway requirements. Consider a gateway application with the following parameters:

- 2 MSCAN nodes and 2 LIN nodes
- 4 Rx frames (one for each node) with 2 signals each
- Each signal has 1 destination
- 6 Tx frames with 7 bytes of payload each

The memory space required to hold all the descriptors can be calculated like this:

- 4 node descriptors (12 bytes each) and 2 LIN node descriptors (18 bytes each) = 84 bytes
- 4 Rx frame descriptors (10 bytes each) and 8 signal descriptors (2 bytes each) = 56 bytes
- 8 signal destination descriptors (2 bytes each) = 16 bytes
- 6 Tx frame descriptors (8 bytes each) and data buffers (42 bytes) = 90 bytes
- 4 transmission queues for 6 Tx frames and 2 LIN Rx frames = (4+6+2)*2 = 24 bytes

The memory space required to hold the gateway database (in this case) is 270 bytes.

5 Generating the Descriptors

Generating the gateway database data that describes the application's desired behavior is laborious (especially for larger systems close to real world requirements). An Excel®[1] spreadsheet with Visual Basic®, macros was created to simplify this process. This section describes how the Excel® spreadsheet can generate the gateway database.

5.1 Generated Files

The macros within the Excel® spreadsheet generate three source files:

- gateway_vector_pointers.h — Contains indexes into the *NodeDescrs* array for the individual on-chip peripherals. These indexes pass the correct node descriptors to the XGATE interrupt service routines.
- gateway_data_dims.h — Contains dimensions of the different data tables. It specifies the total number of Rx frames, Tx frames, and nodes. It also specifies the index of the first LIN node in the *NodeDescrs* array and the number of defined LIN nodes.

1. Excel and Visual Basic are registered trademarks of Microsoft Corporation in the United States and/or other countries.

- gateway_data.cxgate — Contains the different arrays forming the gateway database (Tx frame descriptor array *TxTable*, Rx frame descriptor array *RxTable*, signal descriptor structure *SignalDescrs*, LIN node descriptor array *LinNodeDescrs*, and node descriptor array *NodeDescrs*).

After these source files are generated, they are compiled as part of the gateway example project.

5.2 Node Data Worksheet

The first worksheet with gateway-related data is named Node Data. It defines all the nodes the gateway application will use. Figure 29 shows an example of this worksheet.

	A	B	C	D	E	F	G	H	I	J
2		Ex.: 1 = SCI1/CAN1		[-]	[-]	[bytes]	[-]	[-]	[bytes]	[bytes]
3										***214***
4	0	0	MsCan	1	1	7	2	3	51	
5	1	1	MsCan	1	1	7	2	4	53	
6	2	1	Lin	1	1	8	1	1	66	
7	3	2	Lin	1	0	2	0	0	44	
8										
9										
10										

Figure 29. Example of Node Definitions

Start on line 4 and fill in the first three columns of this worksheet.

Column A contains the node numbers. The node numbers must start at zero and increment by one on each subsequent line. The first blank cell encountered in column A terminates the node list.

Column B identifies the hardware peripheral number used by the node. For example, 0 signifies SCI0 (in case of a LIN node) or MSCAN0 (in case of a CAN node). Similarly, 1 signifies SCI1/MSCAN1, etc.

Column C determines the node type. Only two types of nodes are permitted in the current version of the gateway example code. These are Lin for LIN nodes and MsCan for CAN nodes.

The order of the hardware peripherals in the list is not important (that is, SCI0 can precede SCI1 or the other way round). However, LIN nodes must always follow CAN nodes (that is, CAN node numbers must be lower than LIN node numbers). The descriptor search algorithm requires this order.

Values in the remaining columns of the worksheet are calculated by the macros during the file-generation process and contain statistics about the individual nodes. For example, data sizes required by the individual nodes are calculated in column I and a total data size for the whole gateway application is visible in cell J3.

5.3 Rx Frames Worksheet

The second worksheet is named Rx Frames. It contains all frame definitions the gateway application should receive, definitions of all the signals contained in these frames, and their destinations. Figure 30 shows an example of this worksheet.

The Rx frame data are filled in starting on line 4.

Column A contains the Rx frame numbers. The Rx frame numbers must start at zero and increment by one on each subsequent line. The first blank cell encountered in column A terminates the Rx frame list.

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2	FRM #	Rx ID (hex)	Rx Node	Msg Size	Rx Timer	Rx Timer	Rx Timer	Fail bit	Fail bit	Timeout Handler	byte copy	# of signals
3				[bytes]	prescaler	reload	initial value	enable	position	counter reload		
4	0	20	0	8	20	25	25					2
5	1	21	1	8	1	50	50	X	8	2		2
6	2	29	2	2							X	1
7												
8												
9												
10												

	A	M	N	O	P	Q	R	S	T	U	V	W	X
1		signal #0				signal #1				signal #2			
2	FRM #	position	length	# of dests	destins	position	length	# of dests	destins	position	length	# of dests	destins
3					frame#,position;frame#,position;...								
4	0	1	8	2	0,0;1,15	13	3	1	2,16				
5	1	5	8	3	1,0;0,8;2,0	15	1	1	2,24				
6	2	0	2	1	2,0								
7													
8													
9													
10													

Figure 30. Example of Rx Frame Definitions

Column B holds the received frame's ID, and column C holds the node number the frame is received through. CAN Rx frame descriptors are identified based on the received frame's ID. The search algorithm requires that Rx frame definitions are sorted by node number then ID (both in ascending order).

Column D holds the expected data payload size for the frame. Frames received with shorter-than-expected payload generate an error during the reception process.

Columns E, F, and G hold the Rx timer prescaler, reload value and initial counter value (respectively). The values allowed for the prescaler are restricted to 1, 2, 10, and 20. The reload value and the initial counter value are restricted to a range between 0 and 255. The initial counter value is used to build LIN schedules. All LIN frames belonging to the same schedule are assigned the same reload value (schedule period) and the initial counter value serves as an offset from the beginning of the schedule period. Leaving the cells empty disables the Rx timer.

Non-empty cells in column H enable the fail bit functionality (Section 2.3.1, "Reception," and Section 3.2.5, "Handling of Rx and Tx Timers"). Column I defines the fail-bit position within the Tx frame. The number in column J governs how many timeout periods must elapse before the long timeout is detected.

Column K determines whether a bit-accurate or a byte-accurate-copy algorithm is used. If a non-blank cell is detected, the signals are copied with byte accuracy.

Column L holds the number of signals contained within the frame. This is the last column belonging to the Rx frame descriptor in the *RxTable* array. The remaining columns belong to the signal descriptor structure *SignalDescrs*.

5.3.1 Signal Descriptions

Each signal is described in four columns. Columns M,N,0, and P describe signal #0. Columns Q – T describe signal #1, etc.

The first column (that is, column M for Signal #0) defines the signal position in the Rx frame. The second column (that is, column N for Signal #0) defines the signal length. The third column (that is, column O for Signal #0) stores the number of destinations the signal should be copied into. The last column (that is, column P for Signal #0) stores a string describing the signal destinations.

The signal destination description string contains number pairs, one for every destination. Semicolons separate number pairs (there is no semicolon after the last pair). Commas separate the numbers within the pairs. The first number is the Tx frame number the signal is to be copied into. The second number is the position within the Tx frame where the signal copy should be placed.

5.4 Tx Frame Worksheet

The last worksheet containing the gateway database data is named Tx Frames. It contains definitions of all frames the gateway application will transmit. Figure 31 shows an example of this worksheet.

The Tx frame data are filled in starting on line 4.

	A	B	C	D	E	F	G	H	I
1									
2	FRM #	Tx ID (hex)	Tx Node	Data Size	Tx Timer	periodic Tx	Periodic Tx	Tx on Rx	Tx on data chg
3					prescaler	reload	initial value		
4	0	30	0	7	1	100			
5	1	105	1	7					X
6	2	33	2	8	10	5	5	X	
7	3	47	3	2	20	1	1		
8									
9									
10									

Figure 31. Example of Tx Frame Definitions

Column A contains the Tx frame numbers. The Tx frame numbers must start at zero and increment by one on each subsequent line. The first blank cell encountered in column A terminates the Tx frame list.

Column B defines the transmitted frame's ID, and column C identifies which node the frame will transmit through.

Column D specifies the data payload size of the frame.

Columns E, F, and G hold the Tx timer prescaler, reload value, and initial counter value (respectively). The values allowed for the prescaler are restricted to 1, 2, 10, and 20. The reload value and the initial counter value are restricted to range between 0 and 255. The initial counter value is used to build LIN schedules. All LIN frames belonging to the same schedule are assigned the same reload value (schedule period), and the initial counter value serves as an offset from the beginning of the schedule period. Leaving the cells empty disables the Tx timer.

Non-blank cells in column H specify the frame should be scheduled for transmission immediately after any of the signals it contains are received. Non-blank cells in column I specify the frame should be scheduled for transmission immediately after any of the signals it contains are received, providing the value of the received signal differs from the signal value which was previously transmitted. See Section 2.3.2, "Transmission."

5.5 Generating the Source Files

The source files containing the gateway database can be generated after the worksheets are filled with data describing the desired gateway behavior. The source files are generated by pressing the Generate Files button in the Node Data worksheet. If the source files already exist, they are overwritten.

Very little data consistency and validity verification is performed as part of the file-generation process. You must ensure the data values are valid.

6 References

1. MC9S12XDP512 Data Sheet, Freescale Semiconductor Inc., 2005.
2. Application note AN2732: “Using XGATE to Implement LIN Communication on HCS12X”, Freescale Semiconductor Inc., 2004.

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3333
Rev. 0
11/2006

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

freescale™
semiconductor