Freescale Semiconductor
Application Note

Document Number: AN2732
Rev. 0, 05/2004

Using XGATE to Implement LIN Communication on HCS12X

By Daniel Malik
8/16-Bit Products Division
East Kilbride, Scotland

Contents

1 Introduction . . . 1
1.1 Basic Properties of XGATE . . . 1
1.2 LIN . . . 2
2 Software Structure . . . 2
2.1 Functions . . . 2
3 Data Representation . . . 3
3.1 Data Structures . . . 3
3.2 Schedule Tables . . . 3
4 The Transmit and Receive Driver . . . 4
5 Timeout Management . . . 8
5.1 Timeout Detection Accuracy . . . 8
5.2 Description of the Algorithm . . . 10
6 The Scheduler . . . 11
6.1 Changing the Schedule During Run-time . . . 13

1 Introduction

This application note describes one possible implementation of low level drivers, timeout management and scheduler functions needed for the creation of a master node for LIN communication on the HCS12X family of microcontrollers (MCU).

There are two major classes of possible implementations on this MCU family. The communication functions can be implemented either by using the MCU core or by making use of the XGATE peripheral co-processor. This application note describes an implementation of the second kind that uses the XGATE module.

1.1 Basic Properties of XGATE

The XGATE module is a peripheral co-processor that allows autonomous operation using on-chip RAM and peripherals, with zero load on the main MCU's core.

The XGATE module is an event driven RISC core machine. It has its own instruction set and runs its own

© Freescale Semiconductor, Inc., 2004. All rights reserved.

freescale™ semiconductor

code. The code and data for the XGATE module are stored in the on-chip RAM. Memory sharing is the main method for exchanging data between different threads running on XGATE and also between threads running on XGATE and the MCU's core. From a user's perspective, the HCS12X family devices appear to be a multi-processor environment and hardware semaphores are provided for synchronization of tasks and resource management between threads running on different cores.

The XGATE executes its threads in response to events. These events are issued by the interrupt module, based on its configuration and signals from the on-chip peripherals and MCU core.

1.2 LIN

Local Interconnect Network (LIN) is a low-cost/low-speed automotive network. It allows single master and multiple slave devices to achieve a signal based communication. The individual frames, each consisting of up to eight data bytes, are exchanged using what is known as a "schedule table". This table is stored in the master node, and the master initiates all communication in accordance with the table. The master node may have several different schedule tables and switch between them. The schedule table specifies the identifier for the frame to be exchanged, and the interval between the start of the current frame and the start of the next frame (each frame is allocated a time slot).

2 Software Structure

2.1 Functions

The software required to achieve LIN communication can be structured to form three independent components:

1. Low level Transmit/Receive driver
   This driver serves the individual serial communications interface (SCI) channels used to implement the LIN hardware interface. Due to the special nature of the XGATE signal handling architecture, only one instance of the driver is required, even if there is more than one LIN node implemented on a single chip.
2. Timeout management
   This function is called periodically, based on events generated by one of the on-chip timers, and checks for timeout conditions on any of the LIN channels.
3. Scheduler
   This function manages the schedule table and initiates the exchange of frames. Similarly to the timeout management routine, it is called periodically, based on events from one of the on-chip timers.

3 Data Representation

3.1 Data Structures

Before the above functions can be implemented, structures that describe LIN nodes and individual frames transferred on the bus must be specified.

1. The frame descriptor specifies the memory footprint of one member of the schedule table. It contains the specification of the time slot dedicated to the frame, the frame contents (ID, data length and the data itself), the direction of the frame (transmitted by the master or transmitted by a slave), and a pointer to the next frame descriptor in the schedule table.

```
typedef struct LINframe {
    struct LINframe* next_frame;
    tU16 time;
    tU08 data[8];
    tU08 Id;
    tU08 dir:1;
    tU08 len:4;
} tLINframe;
```

2. The LIN node descriptor provides a pointer to the SCI peripheral used by the node, a pointer to the currently processed frame in the schedule table, a timer variable for timing of the time slot of the current frame, storage space for the frame currently being transmitted or received (ID, data length, the data itself and the checksum), the current state of the Rx/Tx state machine, and a timer variable for timeout detection.

```
typedef struct {
    tSCI* pSCI;
    tU16 checksum;
    tLINframe* frame;
    tU16 frame_time;
    tU08 data[8];
    tU08 Id;
    tU08 dir:1;
    tU08 len:4;
    tU08 state;
    tS08 timer;
} tLINnode;
```

3.2 Schedule Tables

A schedule table is created as a ring of frames, each pointing to the next frame. The schedule function will then go around the ring and transfer the individual frames within their dedicated time slots. An example of such a circular arrangement with four frames is shown in Figure 1.

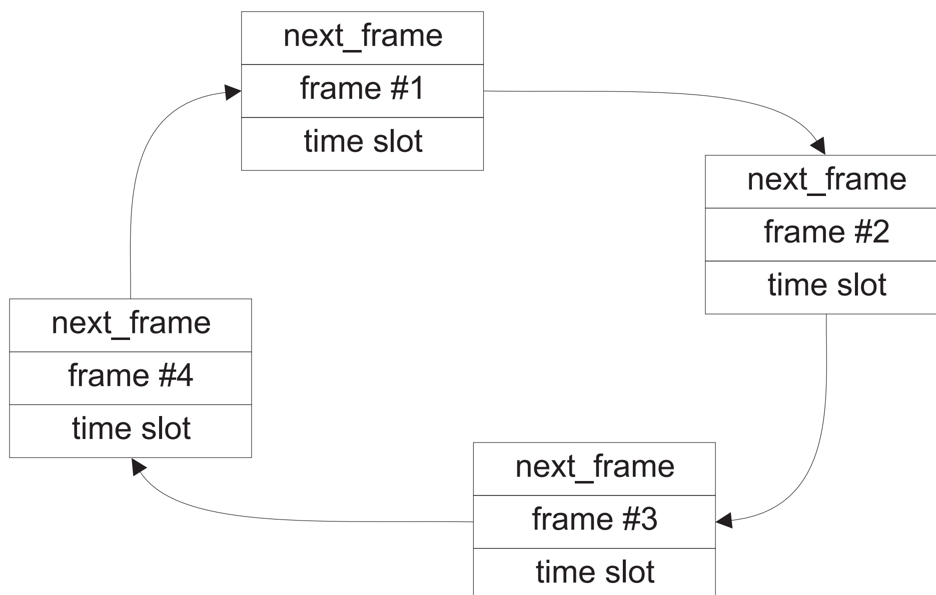

Figure 1. Example of a Schedule Ring Containing Four Frames

4 The Transmit and Receive Driver

The HCS12X family of MCUs offers enhanced serial communication interfaces that support the extensions required to achieve communication in a LIN network (such as collision detection, extended break, and wake-up). However, the interface supports transmission or reception at byte level only and does not provide abstraction layers at frame level. This means that the frame layer must be handled in a software driver responding to the SCI interrupt events.

Since it is impossible to distinguish one byte from another within the LIN frame just by looking at the individual bytes, the transmit and receive driver must implement a state machine that helps to identify which byte of the frame is currently being transmitted or received. This state machine is described by a diagram shown in Figure 2. As the figure is a high level functional description rather than an exact flowchart, the individual actions do not correspond exactly to the states of the state machine; names of the individual states are written above each action description.

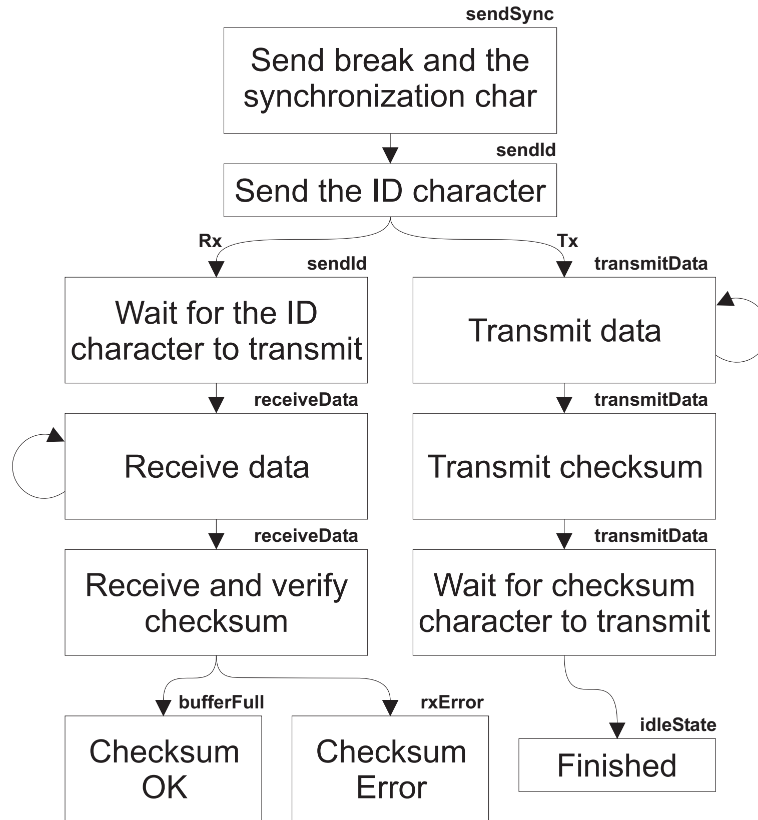

Figure 2. Operation of the Transmit and Receive Driver

Exchange of a LIN frame is started by the scheduler (described later), by changing the state from "idleState" to "sendSync" in the LIN node descriptor, and enabling the SCI Tx buffer empty interrupts. This causes the transmit and receive driver to be executed, and the transfer of the frame is started. The transmit and receive driver finishes in the "idleState" again, in the case of transmission, and in "bufferFull" state, in the case of reception. If an incorrect checksum is obtained during reception, the driver enters the "rxError" state. In the event of a bit error occurring on the bus (indicating a collision or a hardware failure), the driver enters the "bitError" state.

The full implementation of the transmit and receive driver is shown in Listing 1.

Listing 1. C Code of the Transmit and Receive Driver

```
void interrupt XgateXmitLINChar(tLINnode *node) {
   node->pSCI->scisr1.byte;   /* this compiles as read of the register contents */
                              /* since the register is volatile (ANSI C) */
                              /* Check for bit errors */
   if (node->pSCI->sciasr1.abit.berrif) {
      /* Bit collision on the LIN bus - inform the core */
      node->state = bitError;
      _sif();
      return;
   }

   switch (node->state) {
      case sendSync:/* Send break & queue sync */
               node->pSCI->scicr2.bit.sbk = 1;
               node->pSCI->scicr2.bit.sbk = 0;
               node->pSCI->scidrl.byte = 0x55;
               node->state = sendId;
               break;

      case sendId:/* Send ID */
                  /* RIE set means waiting for ID to transmit is in progress */
                  if (node->pSCI->scicr2.bit.rie==0) {
                     if (node->dir) {
                        /* transmit data */
                        node->state = transmitData;
                     } else {
                        /* receive data */
                        /* Change to receiver full interrupt & wait for the ID to transmit */
                        /* cannot use TC flag because of propag. delay of the LIN PHY */
                        node->pSCI->scicr2.byte= RIE|TE|RE;
                        node->pSCI->scidrl.byte;/* read receive buffer to clear it */
                     }
                     node->Id = 0;/* Id becomes data pointer */
                     node->checksum = 0;
                     node->pSCI->scidrl.byte = node->Id;/* transmit the ID */
                  } else {
                     if (node->pSCI->scisr1.bit.tc) {
                        /* transmitter empty => ID was received by slaves */
                        node->state = receiveData;
                        node->timer = 7 + node->len;/* set initial timeout value */
                     }
                     node->pSCI->scidrl.byte; /* read receive buffer to clear it */
                  }
                  break;

      case transmitData:/* transmit data */
                  if (node->Id < node->len) {
                     /* Send byte & update checksum */
                     node->pSCI->scidrl.byte = node->data[node->Id];
                     node->checksum += node->data[node->Id];
                     if (node->checksum>>8) {
                        /* this is equivalent to ADDC on an 8-bit machine */
                        node->checksum=(node->checksum+1)&0x00ff;
                     }
                     node->Id++;
```

```
                } else if (node->Id == node->len) {
                    /* Invert & send checksum */
                    node->checksum = ~node->checksum;
                    node->pSCI->scidrl.byte = node->checksum;
                    node->Id++;
                    /* Change to interrupt on checksum transmit complete */
                    node->pSCI->scicr2.byte= TCIE|TE|RE;
                } else {
                    /* transmit of checksum is now complete */
                    node->pSCI->scicr2.byte= TE|RE;
                    node->state = idleState;
                }
                break;

        case receiveData:/* receive data */
                if (node->Id < node->len) {
                    /* Receive byte & update checksum */
                    node->data[node->Id] = node->pSCI->scidrl.byte;
                    /* increase timeout to account for the currently received char */
                    node->timer += 2;
                    node->checksum += node->data[node->Id];
                    if (node->checksum>>8) {
                        /* this is equivalent to ADDC on an 8-bit machine */
                        node->checksum=(node->checksum+1)&0x00ff;
                    }
                    node->Id++;
                } else {
                    /* all characters received - disable timeout */
                    node->timer = XLIN_TIMER_STOP;
                    /* disable SCI Rx interrupt */
                    node->pSCI->scicr2.byte= TE|RE;
                    /* Invert & compare checksum */
                    node->checksum = ~node->checksum;
                    if (node->pSCI->scidrl.byte != (node->checksum&0x00ff)) {
                        /* Incorrect checksum - inform the core */
                        node->state = rxError;
                        _sif();
                        break;
                    }
                    /* end of reception */
                    node->state = bufferFull;
                }
                break;
        case idleState:
                break;
        default:
                break;
    }
}
```

5 Timeout Management

The main function of the timeout management routine is to make sure that all slaves respond within the time slot dedicated to the frame in the schedule table. Another function of the timeout management routine is to ensure that the system recovers from slave failures (i.e. where the slave does not transmit at all when requested or does not transmit the required number of bytes).

On the other hand, frames transmitted from the master to the slave(s) do not have to be checked for timeouts because, in this case, the master is only transmitting and does not wait for any interaction with the slave(s). There is therefore no opportunity for a dead-lock on the master side.

For the purpose of timeout management, the LIN node descriptor contains the timer variable. During frame reception, this variable holds the time within which the slave must respond by sending the next byte, without violating the LIN specification. The time is decremented periodically by the timeout management routine. If the variable reaches zero, it means that the slave did not respond in time and the LIN node is in the timeout condition.

5.1 Timeout Detection Accuracy

Timing details of a LIN frame are shown in Figure 3. The figure shows that the total number of bits for any frame is $Bit_{COUNT} = 44 + n \times 10$.

From this we can derive the nominal time of a frame: $T_{NOMINAL} = (44 + n \times 10) \times Bit_{TIME}$

The LIN specification defines the maximum time of a frame as 140% of its nominal time.

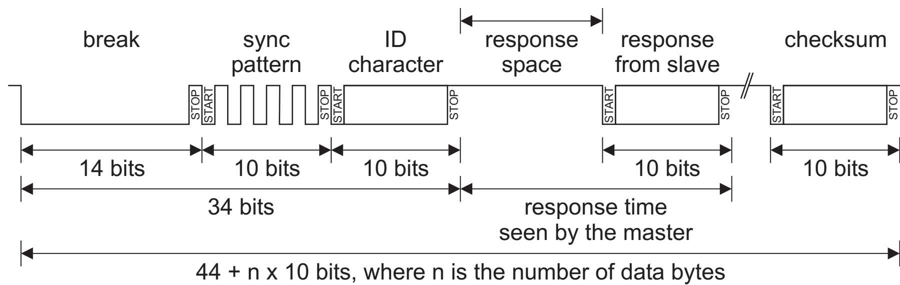

Figure 3. Timing of a LIN Frame

Nominal and maximum times for frames of different lengths can now be calculated based on the above information. The difference between the nominal and the maximum frame time equals the maximum response space the slave can use without violating the specification. While this provides a perfect guideline for detecting the timeout condition, the master cannot detect the response space very easily. It is much easier to implement the timeout management based on measurement of the response time. The results are summarized in Table 1.

Table 1. Calculation of Maximum Response Time

Number of Bytes in Frame	$T_{NOMINAL}$ [bits]	$T_{MAXIMUM}$ [bits]	T_{RESP_MAX} [bits]
0	44	61.6	27.6
1	54	75.6	31.6
2	64	89.6	35.6
3	74	103.6	39.6
4	84	117.6	43.6
5	94	131.6	47.6
6	104	145.6	51.6
7	114	159.6	55.6
8	124	173.6	59.6

The optimum implementation of the timeout management routine depends heavily on the requirements of the application. These requirements are expressed by the schedule table. If the schedule table provides plenty of time for the transfer of each frame, the timeout management can be relaxed, with lower impact on the XGATE loading. However, if the schedule table provides times close to the $T_{MAXIMUM}$ threshold, the timeout management must be more accurate, and this will create a greater load on the XGATE.

The implementation described here is based on the assumption that the schedule table provides at least 160% of $T_{NOMINAL}$ for transfer of each frame. To satisfy this requirement, a resolution of five bit times was selected for the timer. To make the implementation simple the initial load value for the timer variable was selected to be “7 + number of data bytes of the frame”.

The worst case timeout scenarios arise when the number of bytes transmitted by the slave is one less than the expected number (i.e. the slave transmits all data, but fails to transmit the checksum). On the other hand, the timeout detection is fastest (relative to beginning of the frame) if the slave does not respond at all. Table 2 shows when the timeout situation is detected relative to $T_{NOMINAL}$ (measured from the beginning of the frame), in this particular implementation.

Table 2. Time of Timeout Detection Relative to $T_{NOMINAL}$

Number of Bytes in Frame	Timeout Time [5-bit Ticks]	Earliest Timeout Detection [% of $T_{NOMINAL}$]	Latest Timeout Detection [% of $T_{NOMINAL}$]
0	7	145.5	156.8
1	8	127.8	155.6
2	9	115.6	154.7
3	10	100.0	154.1
4	11	94.0	153.6
5	12	89.4	153.2
6	13	85.6	152.9
7	14	82.5	152.6
8	15	75.8	152.4

As can be seen from the table, all worst case timeout detection times are below the "160% of $T_{NOMINAL}$" requirement. Similar timeout detection algorithms can be developed for any particular timing requirement. For example, if the schedule table provides at least 170% of $T_{NOMINAL}$ for transfer of each frame, the timer resolution can be brought down to ten bit times (i.e. the XGATE load created by the timeout management can be halved, in such cases).

5.2 Description of the Algorithm

Once the requirements for the timebase are resolved, the timeout detection itself is uncomplicated, and the function is short. The function loops through all LIN nodes to see if any of them is receiving. The timer variable is decreased for all of the receiving nodes, and a timeout condition is flagged where the timer variable reaches zero. The implementation is shown in Listing 2.

Listing 2. C Code of the Timeout Management Function

```
void interrupt XgateLINErrorPIT(void) {
    register char i;
    PIT.pittf.byte = PTF0;  /* clear the PIT channel 0 interrupt flag */
    for (i=0;i<(sizeof(linp_table)/sizeof(tLINnode*));i++) {
        /* scan through all LIN nodes */
        if (((linp_table[i]->timer)<XLIN_TIMER_STOP)&&((linp_table[i]->timer)>0)) {
            /* timer is within counting range (i.e. the node is receiving) */
            linp_table[i]->timer--;/* decrement the timer */
            if ((linp_table[i]->timer)<=0) {
                /* timeout has occurred - change state of the node and inform core */
                linp_table[i]->state = rxTimeout;
                _sif();
            }
        }
    }
}
```

6 The Scheduler

The scheduler uses the transmit and receive driver to transfer frames over the LIN bus, based on information stored in the schedule rings. The timebase resolution used for the scheduler in this particular implementation is 1 ms. The remaining time of the time slot dedicated to the frame being transferred is stored in the frame_time variable of the LIN node descriptor. The variable is sixteen bits wide, and the maximum time slot dedicated to a frame is therefore 65.535 seconds.

The scheduler loops through all LIN node descriptors and decrements the frame_time variable in each of them. If the time reaches zero, the scheduler advances the frame pointer to the next frame in the schedule ring belonging to the particular LIN node and initiates transfer of the frame. The scheduler also copies data from the LIN node descriptor back to the frame descriptor after the expected number of bytes is received from the slave (receive direction only).

A semaphore is locked before accessing the buffer for either reading or writing, to ensure that the data buffer in the frame descriptor always contains a valid set of data. Also, locking the same semaphore in the code of the application during accesses to the buffers ensures that a mixture of old and new data is never used by the application or by the scheduler.

The implementation is shown in Listing 3.

Listing 3. C Code of the Scheduler Function

```
void interrupt XgateLIN1msPIT(void) {
   register unsigned char *src, *dest;
   register char i;
   PIT.pittf.byte = PTF1;/* clear the PIT channel 1 interrupt flag */
   /* lin scheduler */
   for (i=0;i<(sizeof(linp_table)/sizeof(tLINnode*));i++) {
      if (linp_table[i]->frame!=0) {
         /* there is a frame to be transferred */
         if (linp_table[i]->frame_time>0) {
            linp_table[i]->frame_time--;
         } else if (linp_table[i]->state == idleState) {
            /* timeout occurred & endpoint is idle -> advance to next frame in the schedule */
            linp_table[i]->frame=linp_table[i]->frame->next_frame;
            /* update the time slot for this frame */
            linp_table[i]->frame_time=linp_table[i]->frame->time;
            /* copy data direction and length */
            linp_table[i]->len=linp_table[i]->frame->len;
            linp_table[i]->dir=linp_table[i]->frame->dir;
            /* copy data if transmitting */
            if (linp_table[i]->dir==1) {
               src=linp_table[i]->frame->data;
               dest=linp_table[i]->data;
               _ssem_wait(LINSEMXG);/* wait for semaphore to lock the buffers */
               *(((unsigned int *)dest)+0)=*(((unsigned int *)src)+0);/* data[0] & data[1]
*/
               *(((unsigned int *)dest)+1)=*(((unsigned int *)src)+1);/* data[2] & data[3]
*/
               *(((unsigned int *)dest)+2)=*(((unsigned int *)src)+2);/* data[4] & data[5]
*/
               *(((unsigned int *)dest)+3)=*(((unsigned int *)src)+3);/* data[6] & data[7]
*/
               _csem(LINSEMXG);/* clear the semaphore */
            }
            /* copy identifier */
            linp_table[i]->Id=linp_table[i]->frame->Id;
            /* initiate frame transfer */
            linp_table[i]->state = sendSync;
            linp_table[i]->pSCI->scicr2.byte = TIE|TE|RE;
         }
         if (linp_table[i]->state == bufferFull) {/* data received */
            dest=linp_table[i]->frame->data;
            src=linp_table[i]->data;
            _ssem_wait(LINSEMXG);/* wait for semaphore to lock the buffers */
            *(((unsigned int *)dest)+0)=*(((unsigned int *)src)+0);/* data[0] & data[1] */
            *(((unsigned int *)dest)+1)=*(((unsigned int *)src)+1);/* data[2] & data[3] */
            *(((unsigned int *)dest)+2)=*(((unsigned int *)src)+2);/* data[4] & data[5] */
            *(((unsigned int *)dest)+3)=*(((unsigned int *)src)+3);/* data[6] & data[7] */
            _csem(LINSEMXG);/* clear the semaphore */
            /* change state of the endpoint to idleState */
            linp_table[i]->state = idleState;
         }
      }
   }
}
```

6.1 Changing the Schedule During Run-time

Some applications require the schedule to be changed during run-time, based on the state of the application. There are several ways to implement a run-time schedule change in the environment described so far; for example:

- Immediate change of schedule by altering the LIN node descriptor directly. This involves changing the frame variable to point to the new schedule ring. The scheduler function will advance automatically to the next frame in the new schedule ring, after the time dedicated to the frame currently being transferred elapses.
- Change of schedule after a selected frame from the current schedule ring is transferred. In this case the current schedule ring is opened after the selected frame and modified to point to the new schedule. This situation is depicted in Figure 4. In this case, the scheduler function advances automatically to the new schedule without any external modifications to the LIN node descriptor.

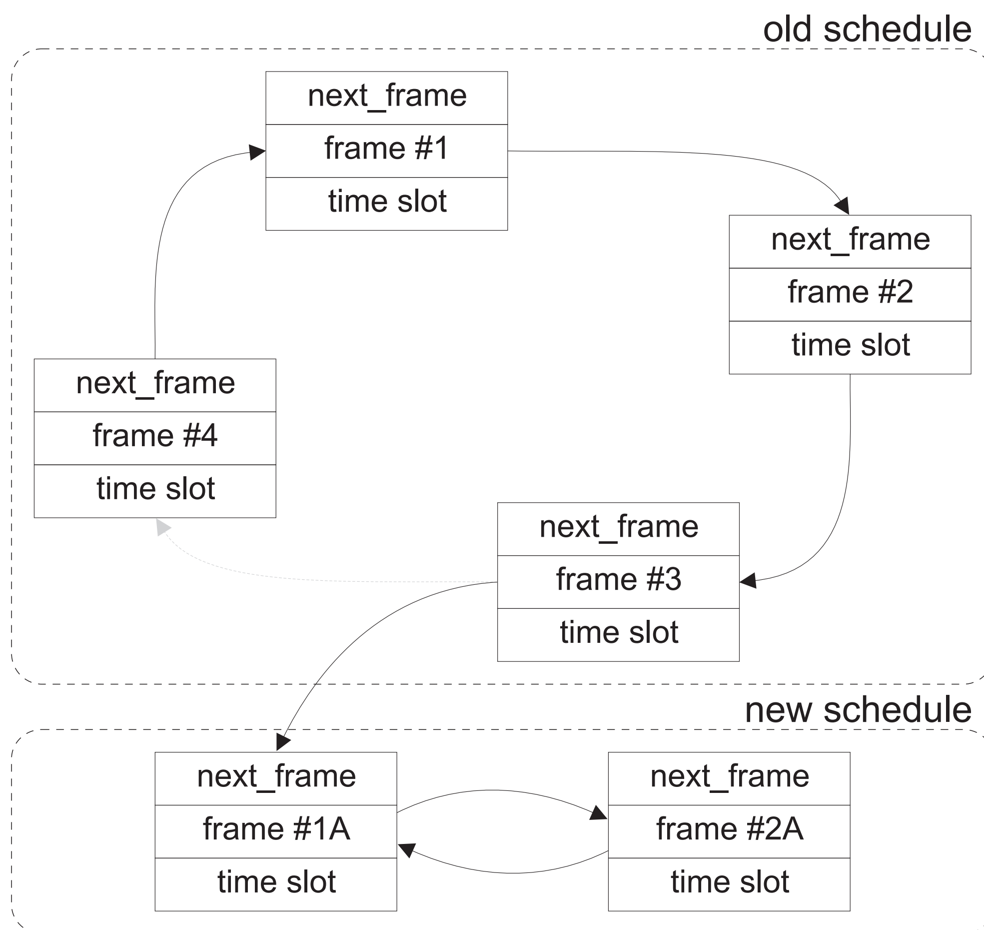

Figure 4. Change of Schedule After a Selected Frame is Transferre

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: AN2732
Rev. 0
05/2004

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2004. All rights reserved.

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