FTT-CAN

Users Guide

Authors:

Paulo Pedreiras
José Alberto Fonseca
Valter Silva

Documentation Version 1, Revision 2, Mai/2004

Documentation only valid for 80C592 and PIC18F458 implementations

1 – Introduction to FTT-CAN protocol.

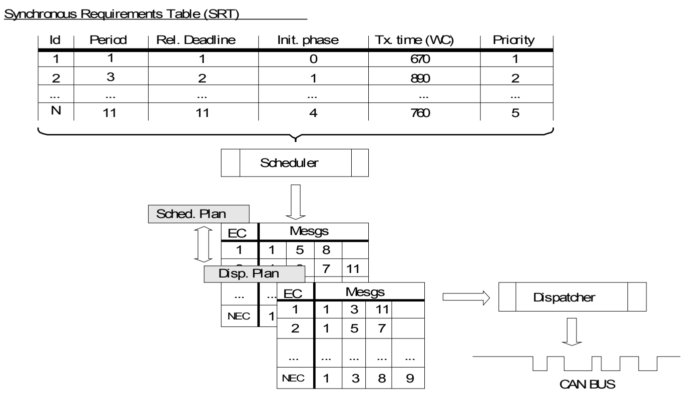
1.1 - Overview

The FTT-CAN (Flexible Time-Triggered communication on CAN) protocol has been briefly presented in [3] and further developed in [4]. A feature that distinguishes this protocol from other proposals concerning time-triggered communication on CAN [2] is that it supports dynamic communication requirements by using centralized scheduling with on-line admission control whilst the communication overhead is kept low by using the native distributed arbitration of CAN.
A Synchronous Requirements Table (SRT) holds the properties of the synchronous message

where:

- DLC : Data length;
- C : Transaction duration;
- Ph : Relative phase;
- P : Period;
- D : Absolute deadline;
- Pr : Priority.

streams:
As usual in table-based scheduling, a finite time resolution in used to express all the properties of the message set. This basic time unit is called Elementary Cycle (EC). The EC duration is fixed and set at pre-run-time. Within each EC, the protocol supports two types of traffic, synchronous and asynchronous. The former one is time-triggered and its temporal properties (i.e. period, deadline and relative phasing) are represented as integer multiples of the EC duration. The latter is transmitted during the periods of the EC not used by the synchronous messages.


Synchronous Requirements Table (SRT)

Id	Period	Rel. Deadline	Init. phase	Tx. time (WC)	Priority
1	1	1	0	670	1
2	3	2	1	890	2
...	...	...	...	...	...
N	11	11	4	760	5

Figure 2 : The planning scheduler

A particular node (Master), scans the current plan and generates a periodic message used to synchronize all other nodes in the network. The transmission of this message represents the start of one elementary cycle (EC) and is known as EC trigger message (TM).

The EC trigger message conveys in its data field the identification of the synchronous messages that must be transmitted by the producer nodes in that EC (EC-Schedule). The nodes that identify themselves as producers by scanning a local table containing the messages to be produced / consumed, transmit the respective synchronous messages in the synchronous phase of that EC (fig. 3). Collisions on bus access are resolved by the native distributed MAC protocol of CAN. This is known as the synchronous messaging system (SMS).

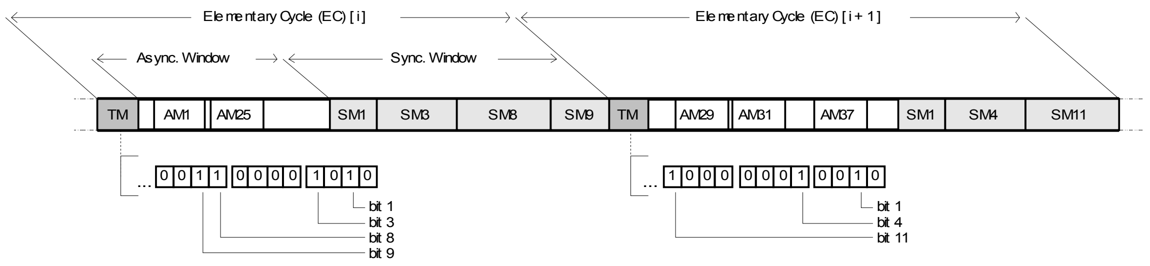


Figure 3 : EC Trigger Message data contents.

The FTT-CAN protocol also supports asynchronous traffic for event-triggered communication, with external control. This sort of traffic is transmitted during the periods of the EC not used by the synchronous messages. However, depending on how the desired temporal isolation between these two sorts of traffic is enforced, the asynchronous messaging system (AMS) can operate in one of two modes [6]. In controlled mode any asynchronous message is transmitted only if it is guaranteed not to interfere with the timeliness of the EC trigger message or of the synchronous messages. In uncontrolled mode, stations wishing to transmit asynchronous messages can try to do it as soon as they receive the respective requests from the application. Although these messages may now cause a certain blocking to the transmission of synchronous ones, such blocking can be upper bounded by using a proper choice of identifiers.

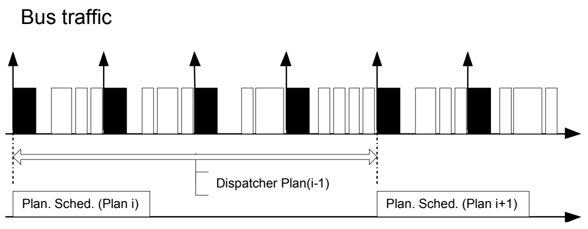


Figure 4 : Scheduler and dispatcher.

In the above picture, it is showed the overall behaviour of the dispatcher and scheduler during one plan. One important aspect that should be noted is that the scheduler, in all invocations, must complete its execution during the dispatching of the current plan. Since the scheduler execution time depends on the specific message set [4], a schedulability test of the scheduler is performed prior to the acceptance of a change request in the synchronous message set.
Thus, a request for a change to the SMS is accepted only and only if both the message set and the scheduler are found to be schedulable. This way, the timeliness of all accepted messages is guaranteed.

1.2 – Message types.

In FTT-CAN there are defined the following message types:

- EC trigger message [MST_MESG_ID], periodically broadcasted by the Master Station, which codifies the periodic messages that should be produced in each EC, as well as the duration of the synchronous length;
- Synchronous Data Messages [DATA_MESG_ID], produced by the stations in the synchronous window, after request of the Master node. These messages convey data to be transferred between stations;
- Asynchronous Data Messages [AM_DATA_MESG_ID], produced by the stations in the asynchronous window, under a best effort policy. These messages convey data to be transferred between stations;
- Control Messages [CONTROL_MESG_ID], produced in the asynchronous window. These messages are used for systemm management, allowing:
    - Synchronization among Masters
    - Change requests to the message set

The four most significant bits of the ID field [ID.b10...ID.b7] of the trigger message codify the message type according to the following rules:

0 [Synch]	0 [MASTER]	00	Trigger Message

	1 [SLAVE]	10	Data Message
		00,01,11	Not used

1 [ASYNCH]	000	Control Message [HP]
	100	Data Message [RT]
	110	Control Message [LP]
	111	Data Message [NRT]
	001,010,011,101	Not Used

Figure 5 : Message type identification

1.3 The Trigger Message

The TM is transmitted by the active Master at the beginning of each EC and conveys:

- •The master identification
- •A sequence number
- •The duration of the Sychronous window
- •A bitmap indicating which synchronous messages shall be produced in the EC (EC—Schedule)
- •A flag signaling the beginning of a new plan

Its structure is depicted in the following table.

Type	***Master ID***	***New Plan***	***Sequence number***	***Synch. Window length***	***Message Trigger Field***
[0001]	[0..7]	[0/1]	[0..7]	[0..255]	Bitmap
CAN ID field				CAN Data Field	
b10...b7	b6...b4	b3	b2...b0	MSB	1 to 7 bytes

Table 1: Trigger Message structure

The synchronous window length is unitless, and indicates the quantity of SWL_SLOTS used by the synchronous window. This parameter is defined during system design, and depends on the length of the elementary cycle.

The “New Plan” flag is set by the Master whenever the EC is the first one of a new plan. This field is used to facilitate the master synchronization.

The sequence number is incremented in every EC. This field allows all the nodes in the system to detect up to 8 consecutive missing TMs.

The Message Trigger field carries a bitmap indicating which synchronous messages should be produced in the EC, .i.e, the EC-Schedule. The number of data bytes is fixed during run-time, and thus the TM always has the same length, independently of the quantity of messages being transmitted in each EC. The codification of the message IDs is performed in ascending order, right-to-left.

...		***Data Byte [1]***					***Data Byte [0]***				
...	16	15	...	10	9	8	7	...	2	1	0

1.4 Synchronous Data Message

The synchronous data messages are used to periodically distribute state data among the network nodes, and are always transmitted within the synchronous window, when indicated in the trigger message. The synchronous data message structure includes:

- The message identification
- A TX new data field (TX_ND)
- The data itself

Its structure is depicted in the following table.

Type	***TX_ND***	***Message ID***	***Data field***
[0110]	[0/1]	[0..64]	Application dependent
CAN ID Field			CAN Data Field
b10...b7	b6	b5...b0	1 to 8 bytes

Table 2: Synchronous Data Message structure

The synchronous data messages are transmitted autonomously by the communication system, without explicit intervention of the application. Therefore it is usefull to have information about the freshness of data carried in the message. The TX_ND flag, if set, indicates that the source node has updated its local image of the respective real-time entity after the last transmission. Conversely, if this bit is not set, it means that the application had not updated the local image, and thus the contents of the message is the same as as the one in its last intance.

1.5 Asynchronous Data Message

Asynchronous data messages are used to convey event information and are sent after application explicit request. Asynchronous data messages are always transmitted within the asynchronous window. Since there is no global knowledge about how many asynchronous messages are ready for transmission in the beginning of each asynchronous window, each station is responsible by verifying near the end of the window if the message can still be transmitted. If not, the message must be removed from the controllers transmision buffer and placed in the local queue again. This process is repeated in all Ecs until the message is sucessfully transmitted. The arbitrartion among asynchronous messages relies on the native CAN MAC, and thus uses the ID field. The structure of a asynchronous data message is similar to its synchronous siblin, except that there is no TX_ND field.

- The message identification

- The data itself

Its structure is depicted in the following table.

Type	***Not used***	***Message ID***	***Data field***
[1100] (RT) [1110] (NRT)	---	[0..64]	Application dependent
CAN ID Field			CAN Data Field
b10...b7	b6	b5...b0	1 to 8 bytes

Table 2: Asynchronous Data Message structure

There are defined two types of asynchronous data messages, the real-time (RT) and the non-real-time (NRT). The RT ones are subject to pre-runtime analysis, and thus it can be verified in advance if its timeliness requirements will be fulfiled. NRT asynchronous messages are handled under a best-effort policy, therefore no timeliness guaranties can be satisfied. Since the most significant bits of the RT message IDs are lower (higher priority) for RT than for NRT messages, the CAN MAC arbitration selects the RT messages to be scheduled before the NRT ones.

1.6 Asynchronous Control Messages

Asynchronous control messages are used to convey system related data, such as master synchronization data, software download, requests for SRT changes, etc. The ID field of all the control messages is the same.

•Control message identification

•Control message data

Its structure is depicted in the following table.

Type	***Not used***	***Message ID***	***Data field***
[0000] (HP) [1111] (LP)	---	[0..64]	Application dependent
CAN ID Field			CAN Data Field
b10...b7	b6	b5...b0	1 to 8 bytes

Table 2: Control Message structure

As for the case of asynchronous   data messages, there are two types of control messages, high and low priority. The high-priority messages have the highest priority among all the asynchronous messages, and thus should be used just for time-critical operations. The lower priority messages have the lower priority among all the asynchronous messages. These shouild be used to carry operations that are not time constainted, such as remote diagnosis, logging, etc..

1.7 Example: decoding the TM

It follows the pseudo code for the implementation of TM decoding and data message transmission on a FTT-CAN station.

```
/***************************************/
/* CAN Rx INT : message just received */
/***************************************/
if( interrupts & RI_MASK)     /* RI - Rx interrupt*/
{
        /* Read the full frame of the master message into IntMesgBuf[]  */
        CAN_Get_Frame(IntMesgBuf);

        /* Release RX buffer */
        CAN_Release_Buffer();

        /* Message read. Check message type */

        /***************************/
        /* If Trigger Message, ... */
        /***************************/
        if( (IntMesgBuf[0] & TM_MASK )
        {
```

```
        /* Check current phase (if asynchronous window -> duplicated TM */
        /*    -> Discard message*/
        if(win_phase == ASY_WIN)
                return; /* Duplicated TM – do nothing */

        /* Get Asynch Window Duration from the TM */
        swlen_us = IntMesgBuf[2]*SWL_SLOT_us;

        /* Set timer: subtract the protocol overheads from the EC duration */
        aw_time= LEC - (swlen_us + INT_CAN_OVERH_us + TM_us + MAXAMVAR_us + \
                        OVRHEAD_QUEU_us);
        Timer(aw_time);


        /* Set-up of status variables */
        win_phase = ASY_WIN; /* Current phase is asynchronous window */
        AM_stop = FALSE;     /* Signals that Asynchronus Messages are allowed */

        /* Increment EC_counters */
        SEC_count++;
        AEC_count++;

        /***************************************************/
        /* Prepare production of synchronous messages        */
        /* Fill IntTxBuf with first message, update counters */
        /***************************************************/

        /* Initialise EC message counters */
        NEcTxMesg= 0;
        ScanIndex= 0;
        NEcRxMesg= 0;


        /* Check for messages locally produced */
        for( mid = 0; mid < StNProdMesg; mid++)
        {
                if( CheckProducer(IntMesgBuf,mid )
                { /* If message locally produced ...*/
                        Update_TXBUF(mid); /* Copy message to TxBuf: */

                        SRT[mid].status &= ~TX_NEW_DAT; /* Reset TX NEW_DAT flag*/

                        /* Save production buffer indexes */
                        /* NOTE: remaining messages will be txmited after TX Int */
                        ScanIndex = mid;
                        NEcTxMesg = 1;
                        break;


                } /* If bit set in Trigger Message */
        }  /* For each message in SRT */
} /* IF Trigger Message */

/**********************************/
/* If Synchronous Data Message... */
/**********************************/
```

```
        else if( IntMesgBuf[0] & SDATA_MESG__MASK )
        {   /* Check if locally consumed var */

               mid=GetId(IntMesgBuf); /* Get mesg ID field */

               if( CheckConsumer(IntMesgBuf,mid,SYNC)
               { /* Update local image of RT entity */
                   Update_Buff(IntMesgBuf,mid);
               } /* If Consumer */

        } /* Synchronous data message received */

        /**********************************/
        /* If Asynchronous Data Message.. */
        /**********************************/
        else if( IntMesgBuf[0] & AM_DATA_MESG_MASK )
        {   /* Check if locally consumed message */

               mid=GetId(IntMesgBuf); /* Get mesg ID field */

               if( CheckConsumer(IntMesgBuf,mid,ASYNC)
               { /* Update local image of RT entity */
                   Queue_Message(IntMesgBuf,mid);
               } /* If Consumer */

        } /* Asynchronous data message received */

} /* RX Int */
```

2- System Setup.

Current implementation of FTT-Can protocol is made on CANivete boards. This hardware is based on the Philips 80C592 microcontroller. Detailed information on software libraries and hardware platform can be found in [7] and [8].

Figure 5 presents a typical system based on CANivete boards running FTT_CAN.

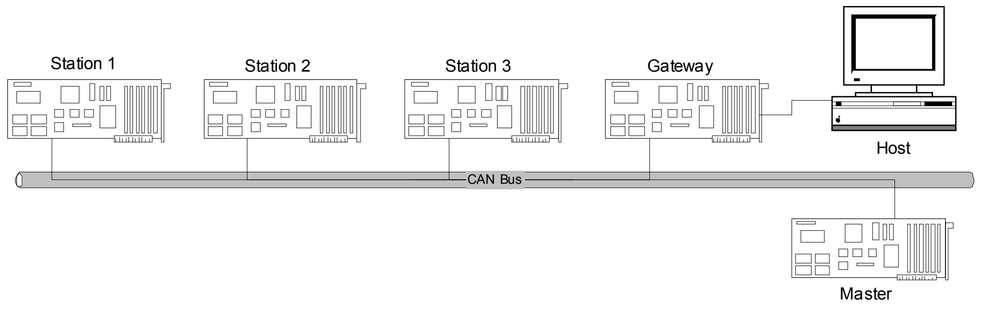


Figure 5 : Generic system configuration.

Each of the board has an identifier stored in the EPROM that must be unique in the system. The Gateway board has always the identifier 0, and its function is transporting messages sent from the PC (via serial port) to the CAN bus. The remaining boards, after power-up or “Reset 0” stay in remote programming mode. In this mode, programs can be uploaded from the host PC to each one of the boards via the gateway.
Once a node has a program loaded, it can be set in execution mode pressing the “Reset 1” button. To replace the program loaded in one board, the “Reset 0” button should be pressed, and the load procedure repeated.

The communication system code is hidden from the user, and all the interactions are performed via an interface, which consists of a set of state variables and functions. These state variables and interface functions, as well as some specific data structures are packed in a library that must be included in all the programs that require the use of FTT-CAN communications. The corresponding header is “fttcan8.h”. The data structures, functions and state variables are described in following sections 3 and 4.

To set-up a system, the user must perform the following steps:

i)Define the set of messages (and respective properties) that will be carried by the Network;

ii)Assign messages to nodes;

iii)Write the code for the Master (see section 3) and Stations (see section 4);

iv)Download the code in the desired nodes. The host must be connected to the Gateways serial communication interface thru a serial cable;

v)Press the “Reset 1” in each board. Note that the stations only will transmit messages after the Master node starting broadcast Trigger Messages.

3 – The Master node.

3.1 - Overview

The Master node has the role of building a schedule based on the requirements of the current set of synchronous messages. Based on this schedule, it coordinates the exchange of messages on the bus. This coordination is achieved since all stations only produce the synchronous messages when authorized by the EC trigger message. As the Scheduler implemented in FTT-CAN is “Planning based”, the schedule contains the messages that should be exchanged during a limited amount of time – the Plan. Consequently the Scheduler must be executed periodically to build another schedule. To avoid conflicts between the Dispatcher and the Scheduler, while the former processes one plan, the latter builds the next one using another data structure. When the Dispatcher finishes processing one plan, the data structures are exchanged, and the Scheduler can be executed again.

3.2 – Data Structures.

The Master holds data structures where the properties of each message are stored. For the synchronous messages (Synchronous Requirements Table – SRT), the data is organized as follows:

```
typedef struct
{
    unsigned char vid;         /* variable's id (lower 6 bits) */
    unsigned char byte_index;  /* byte of master mess. data where this var. is codified */
    unsigned char bit_mask;    /* bit within [byte_index] corresponding to var. */
    unsigned char size;        /* # of data bytes */
    unsigned char b_size;      /* size in bits (including stuff bits) */
    unsigned char period;      /* period in # of ECs */
    unsigned char deadline;    /* message deadline, usually=period */
    signed char init;          /* Initial phase (EC of first occurrence) */
    unsigned char status;      /* PRODUCER, CONSUMER, TX_NEW_DAT, RX_NEW_DAT */
    unsigned char max_jit;     /* save max value of jitter (in # Ec's) */
   unsigned char nsec;       /* worst-case response in ECs */
   unsigned char occurmax;   /* max number of occurrences of var within a plan */
} MstVarType;


typedef struct
{
    unsigned char nvars;
    MstVarType vars[MAX_VAR];
} MstVTabType;
```

Some of these variables are filled by the user and others are calculated by the function that appends the data to the table.

The user must supply the following parameters:

- vid : holds the lower 6 bits of the variable id;

- size : data size, in bytes;
- period : message period, in ECs;
- deadline : message deadline, in ECs;
- init : the initial phase of the message, in ECs.

When the message is added to the SRT, the following parameters are calculated:

- b_size : maximum size of the message in bits. Stuff bits included;
- byte_index & bit_mask : location of the bit in the EC trigger message where the production of the message is codified. This operation generates redundant data, because these values are derived from the message id, however it allows improving run-time efficiency.

3.3 - Primitives.

To set-up the master, the user needs to know the properties of the synchronous messages that should be produced. There are the following interface functions:

int FTT_MstInit(void);

This function initialises all the variables in the system. It must be called before any other operation involving the FTT protocol;

signed char MstTab_AddVar(MstVarType * MstNewMesg);

This function adds a message to the message table.

Arguments:

▪MstNewMesg : must be filled with the message properties, as described in 3.2

The return code is:

▪0 : OK

▪–1 : maximum number of messages is reached.

void FTT_MstStart(void);

Starts the scheduling and dispatching of the messages. This must be the last operation performed. Any code placed after the invocation of this function will be ignored.

Below it is the code for set-up a master where 2 synchronous messages should be produced.

```
/****************************************************/
void main(void)
{
        MstVarType aux_var;    /* aux_var is used to pass data to MstTab_AddVar() */
        /* Initialisations - CAN, timer, global variables */
        FTT_MstInit();

        /* Add message 1 : {   Id = 1 ; 8 data bytes ; Period = 1 ; Deadline = 1 ;
                               Initial phase = 1 } */
        aux_var.vid=0x01;
        aux_var.size=8;
        aux_var.period=1;
        aux_var.deadline=1;
        aux_var.init=1;
        MstTab_AddVar(&aux_var);

        /* Add message 2 : {   Id = 2 ; 4 data bytes ; Period = 3 ; Deadline = 3 ;
                               Initial phase = 1} */
        aux_var.vid=0x02;
        aux_var.size=4;
        aux_var.period=3;
        aux_var.deadline=3;
        aux_var.init=1;
        MstTab_AddVar(&aux_var);

        /* Start the scheduling and dispatching */
        FTT_MstStart();
} /* main end */
/*****************************************************/
```

4 – The Station nodes.

4.1 - Overview

The Stations can produce both synchronous and asynchronous/sporadic messages. The synchronous messages are produced according to the Master message broadcasted by the Master node. There is a reserved area for the production of these messages, which is located at the end of the EC (Synchronous Window). The asynchronous messages are produced in the Asynchronous Window, located between the reception of the EC Master message and the beginning of the synchronous window.

The asynchronous window uses all the space not required by the synchronous window, so its length depends on the synchronous load of each EC (Fig. 3). Since in the asynchronous window there is no central coordination, the asynchronous messages are handled under a best-effort policy.

For the sporadic messages, an analytic analysis can be performed if all the minimum inter-arrival times are known [6].

4.2 – Data Structures.

The Station nodes have data structures where the properties of synchronous messages either produced or consumed are stored. Concerning the asynchronous/sporadic messages, only the properties of the locally consumed messages are stored.

4.2.1 – Synchronous messages.

```
/* Station - Synchronous messages */
typedef struct
{
    unsigned char vid;        /* variable's id (lower 6 bits) */
    unsigned char byte_index; /* byte of master mess. data where this var. is codified */
    unsigned char bit_mask;   /* bit within [byte_index] corresponding to var. */
    unsigned char data0[8];   /* double-buffer to hold variable's value  */
    unsigned char data1[8];
    unsigned char size;       /* n. of data bytes */
    unsigned char status;     /* PRODUCER, CONSUMER, TX_NEW_DAT, RX_NEW_DAT, LASTBUF */
} StaVarType;

/* Station synchronous variables table type */
typedef struct
{
    unsigned char nvars;
    StaVarType vars[MAX_VAR];
} StaVTabType;
```

Some of these variables are filled by the user and others are calculated by the function that appends the data to the table.

The user must supply the following parameters:

- vid : holds the lower 6 bits of the variable id;
- size : data size, in bytes;
- status : producer or consumer.

The **status** field contains information about several properties, namely:

PRODUCER: if set, signals that the station is producer of the message;
CONSUMER: if set, signals that the station is a consumer of the message;
TX_NEW_DAT: if set, signals that the value was refreshed at the source before transmission;
RX_NEW_DAT: if set, signals that a new message has been received since last reading of the local buffer;
LASTBUF: used internally; signals which of the buffers contains the most recent data.

This field should be consulted/updated via the masks provided. For instance:

```
…
aux_var.status|=PRODUCER;     /* Set station as the producer of the message */
…
aux_var.status|=CONSUMER;     /* Set station as a consumer of the message */
…
if( (aux_var.status) & TX_NEW_DAT ) /* To check the TX_NEW_DAT flag */
{
        …
}
```

The TX_NEW_DAT and RX_NEW_DAT flags allow to assess the accuracy of a variable in the time domain.
The TX_NEW_DAT flag is managed by the Producer node. It is set to true whenever the local variable’s buffer is updated by the application, and it is set to false upon transmission of the respective message. Therefore, when a message has this flag set to true, it means that the value carried is up to one period old.
The RX_NEW_DAT flag is managed at the Consumer side, and is set to true whenever the variable’s buffer is updated by a network message, and set to false upon consumption by the application. Therefore, this flag is true whenever there is a new (not yet consumed) value in the variable’s buffer. This status is not directly related to the temporal accuracy of the current variable value in the buffer, but can be useful for synchronization purposes between the communication system and the consumer task.

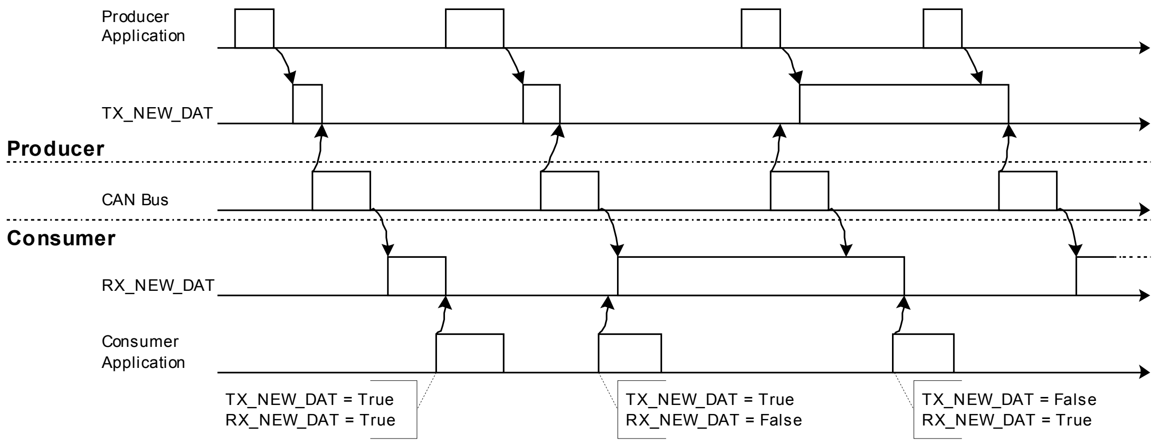


Figure 6 : TX_NEW_DATA and RX_NEW_DATA flags.

When the message is added, the following parameters are calculated:

- byte_index & bit_mask : location of the bit in the EC trigger message where the production of the message is codified. This operation generates redundant data, because these values are derived from the message id, however it allows improving run-time efficiency.

4.2.2 – Asynchronous messages.

```
/* Station - Asynchronous messages to consume type */
typedef struct
{
    unsigned char vid;         /* variable's id (lower 6 bits) */
    unsigned char data0[8];    /* double-buffer to hold variable's value  */
    unsigned char data1[8];
    unsigned char size;        /* n. of data bytes */
    unsigned char status;      /* PRODUCER, CONSUMER, TX_NEW_DAT, RX_NEW_DAT, LASTBUF */
} AM_ConsStaVarType;

/* Station - Asynchronous variables table type */
typedef struct
{
    unsigned char nvars;
    AM_ConsStaVarType vars[AM_STAMAXVAR];
} AM_ConsStaVTabType;
```

The user must supply the following parameters:

- vid : holds the lower 6 bits of the variable id;
- size : data size, in bytes.

As in the case of the synchronous messages, the status field holds some info about the status of the message. At the user level and for this kind of messages, the only relevant flag is:

<u>RX_NEW_DAT</u>: if set (true) signals that a new message has been received since last reading of the local buffer.

4.3 - Primitives.

To set-up the stations, the user needs to know the properties of the synchronous messages that should be produced and consumed by the station, as well as the properties of the asynchronous messages that are supposed to be consumed. The following interface functions are available:

•**General functions:**

void FTT_StaInit(void);

This function initialises all the variables in the system. It must be called prior to any other operation is performed;

void FTT_StaStart(void);

Activates the FTT system. Should be called after message set-up.

•**Synchronous Messages functions:**

signed char StaTab_AddVar(StaVarType* StaNewMesg);

This function adds a message to the synchronous message table.

Arguments:

▪StaNewMesg : must be filled with the message properties, as described in 4.2.1.

The return code is:

▪0 : OK;

▪-1 : table full (var_table.nvars = MAX_VAR);

▪-2 : status not specified (aux_var->status = 0);

▪-3 : vid index does not fit (aux_var->vid >= MAX_VAR);

▪-4 : production buffer full (StNProdMesg = STA_MAX_VAR).

signed char VarStatus(unsigned char MesgID);

This function allows to check the status field of one synchronous message.

Arguments:

▪MesgID : ID of the message.

The return code is:

▪-1 : message not in table;

▪Other : Status field. The individual flags should be obtained through the masks.

signed char produce(unsigned char MesgID, unsigned char * Value);

This function checks weather station is a producer of this message and copies the variable's value into the value buffer in the Station synchronous message table.

Arguments:

▪MesgID : ID of the message to produce;

▪Value : Address of the buffer where the data to be carried by the message is stored.

The return code is:

▪-1 - message not in table;

▪-2 - station is not producer;

▪Other : TX_NEW_DAT flag. The state of this flag should be obtained through the respective mask.

signed char consume(unsigned char MesgID, unsigned char * Value);

This function checks weather station is a consumer of this message and copies the variable's value from the value buffer in the Sta. Var. Table.

Arguments:

▪MesgID : ID of the message to produce;

▪Value : Address of the buffer where the data field of the consumed message should be stored.

The return code is:

▪-1 : message not in table;

▪-2 : station is not consumer;

▪Other : Status field. The individual flags should be obtained through the masks.

•Asynchronous Messages functions:

signed char AM_StaTab_AddVar(AM_ConsStaVarType* AMNewMesg);

This function inserts a variable in the station asynchronous variables table. The table has a fixed size of AM_STAMAXVAR messages. Only variables that are consumed in the station need to be stored.

Arguments:

▪AMNewMesg : must be filled with the message properties, as described in 4.2.2.

The return code is:

▪0 : OK

▪-1 : table full (am_var_table.nvars = AM_STAMAXVAR)

▪-3 : invalid ID (> 6 bits <=> > 63d)

signed char AM_VarStatus(unsigned char MesgID);

This function allows to check the status field of one asynchronous message.

Arguments:

▪MesgID : ID of the message.

The return code is:

▪-1 : message not in table;

▪Other: Status field. The individual flags should be obtained through the masks.

signed char receive (unsigned char MesgID, unsigned char * Value);

This function checks if station is a receiver of the message and copies the variable's value from the value buffer in the Asynchronous Message table.

Arguments:

- MesgID : ID of the message to consume;
- Value : Address of the buffer where the data field of the consumed message should be stored.

Return code:

- < 0 [AMIID] - message not in table;
- Other : Status field. The individual flags should be obtained through the masks.

**signed char send(unsigned char MesgID, unsigned char MesgLen,
unsigned char *Value);**

This function copies a variable's value into the Asynchronous Transmit Buffer for transmission. Checks weather the transmit buffer is full. The message waits on the buffer (eventually for several ECs) until transmission is completed successfully.

Arguments:

- MesgID : ID of the message to produce;
- MesgLen : length (in bytes) of the data field;
- Value : Address of the buffer where the data to be carried by the message is stored.

The return code is:

- AMOK : Success
- AMBF : Buffer full
- AMIID : Invalid Msg ID
- AMIML : Invalid Msg length

Below it is the sample code to set-up a station

```
/***************************************************/
void main(void)
{
    /* Variables used to update Message Tables */
    StaVarType aux_var;
    AM_ConsStaVarType am_aux_var;


    /* Variables' values */
    unsigned char value1[8],value3[3];        /* Synchronous */
    unsigned char am_value1[4],am_value5[2];  /* Asynchronous */
```

```
/* Return code and Status variables */
signed char vstatus;
signed char result;

/* Messages used to produce asynchronous messages */
unsigned char mesgid;
unsigned char mesglen;

/**************************************************************/
/* This set of functions must be called to proper system set-up */
FTT_StaInit();     /* Init global vars and timers */
/**************************************************************/

/*****************************/
/* User codes starts here ... */
/*****************************/

/*  Build variables' table */

/* Synchronous Messages */
/* Message 1 ; Size 8; Producer */
aux_var.vid=0x01;
aux_var.size=8;
aux_var.status=0;
aux_var.status|=PRODUCER;
vstatus=StaTab_AddVar(&aux_var);
if( vstatus < 0 )  /* Checks if OK */
{
    /*… Place error handling code here … */
}

/* Message 3 ; Size 3; Consumer */
aux_var.vid=0x03;
aux_var.size=3;
aux_var.status=0;
aux_var.status|=CONSUMER;
vstatus=StaTab_AddVar(&aux_var);
if( vstatus < 0 )  /* Checks if OK */
{
    /*… Place error handling code here … */
}

/* Asynchronous Messages (station is a consumer) */
/* Message 1 ; Size 4 */
am_aux_var.vid=0x01;
am_aux_var.size=4;
vstatus=AM_StaTab_AddVar(&am_aux_var);
if( vstatus < 0 )  /* Checks if OK */
{
    /*… Place error handling code here … */
}

/* Start the FTT system */
```

```
    FTT_StaStart();

    /* Set-up finished. Control Loop code follows */
    while(1)
    {
        /* Station reads a temp sensor and produces sync. message 1 with its value */
        /*   if a new sample is available. Notice that the effective production of */
        /*   the message will happen only after Master authorization. Production   */
        /*   will occur even if the value is not updated.                          */
        if(Temp_Sensor_New_Data(value1))
            produce( 1, value1);

        /* Station checks synchronous message 3. If new data update actuator. */
        vstatus = VarStatus(3);    /* Get message 3 status */
        if(vstatus > 0)            /* If valid ID */
            if(vstatus & RX_NEW_DAT)   /* Test if a new value was received */
            {
                consume(3,value3);        /* Get the new value */
                Update_Actuator_3(value3);/* Update actuator */
            }

        /* Station reads the status of a switch, and if there is a change in its   */
        /*   status produces asynchronous message 5 with the reading. The message  */
        /*   is placed in the production buffer and will be produced as soon as    */
        /*   possible*/
        if(Switch_Change(am_value5))
        {
            mesgid=5;
            mesglen=2;
            result=send(mesgid,mesglen,am_value5);
            if(result != AMOK)
            {
                /* Place error handling code here */
            }
        }

        /* Station checks asynchronous message 1. If new message was received      */
        /*    update actuator. */
        vstatus = AM_VarStatus(1);/* Get asynch. message 1 status */
        if(vstatus > 0)            /* If valid ID */
            if(vstatus & RX_NEW_DAT)   /* Test if a new value was received */
            {
                receive(1,am_value1);         /* Get the new value */
                Update_Actuator_1(am_value1); /* Update actuator */
            }
    } /* End while(1) */

} /* End main() */

/*****************************************************/
```

Glossary.

AMS	Asynchronous Messaging System
C_i	Transaction duration of message I
FTT-CAN	Flexible Time-Triggered communication on CAN
CAN	Controller Area Network
D_i	Absolute deadline of message i (in ECs)
DLC_i	Data length (in bytes) of message i
EC	Elementary Cycle
P_i	Period of message i (in ECs)
Ph_i	Relative phase of message i (in ECs)
Pr_i	Priority of message i
SMS	Synchronous Messaging System
SRT	Synchronous Requirements Synchronous
TM	Trigger Message

.

References

[1] Kopetz, H. Real-Time Systems Design Principles for Distributed Embedded Applications. Kluwer Academis Publishers, 1997

[2] Peraldi, M.A. and J.D. Decotignie. Combining Real-Time Features of Local Area Networks FIP and CAN. Proc. of ICC’95 (2nd Int. CAN Conference), CiA – CAN in Automation, 1995.

[3] Almeida, L., J.A. Fonseca, P. Fonseca. A Flexible Time-Triggered Communication System Based on the Controller Area Network: Experimental Results. Proc. of FeT’99 (Int. Conf. on Fieldbus Technology), Magdeburg, Germany, September 1999.

[4] Almeida, L. Flexibility and Timeliness in Fieldbus-based Real-Time Systems. PhD Thesis, University of Aveiro, Portugal, November 1999.

[5] Fohler, G. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. Proc. 16th Real-Time Systems Symposium, Pisa, Italy, 1995.

[6] Pedreiras, P., Almeida, L. Combining Event-triggered and Time-triggered traffic inFTT-CAN: Analysis of the Asynchronous Messaging System. Proc. WFCS 2000, Oporto, 2000

[7] Pedro Fonseca. Rotinas para o 80592, Outubro/19997

[8] Grupo de Sistemas Electrónicos Distribuidos – Dep. De Electrónica e Telecomunicações - Universidade de Aveiro. Manual do utilisador do sistema CANivete, Julho /1998