US006647440B1

(12) **United States Patent**
Birns et al.

(10) **Patent No.: US 6,647,440 B1**
(45) **Date of Patent: *Nov. 11, 2003**

(54) **END-OF-MESSAGE HANDLING AND INTERRUPT GENERATION IN A CAN MODULE PROVIDING HARDWARE ASSEMBLY OF MULTI-FRAME CAN MESSAGES**

(75) Inventors: **Neil Edward Birns**, Cupertino, CA (US); **William J. Slivkoff**, San Jose, CA (US); **Hong Bin Hao**, San Jose, CA (US); **Richard Fabbri**, Stamford, CT (US); **Jie Zheng**, Mountain View, CA (US)

(73) Assignee: **Koninklijke Philips Electronics N.V.**, Eindhoven (NL)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 398 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/630,288**

(22) Filed: **Aug. 1, 2000**

Related U.S. Application Data

(60) Provisional application No. 60/154,022, filed on Sep. 15, 1999.

(51) **Int. Cl.**[7] **G06F 13/14**

(52) **U.S. Cl.** **710/29**; 379/133; 370/470; 709/238

(58) **Field of Search** 710/29; 379/133; 370/470; 709/238

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,127,742 A * 11/1978 Couturier et al. ..... 179/18 FC
4,604,682 A 8/1986 Schwan et al. 364/200

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

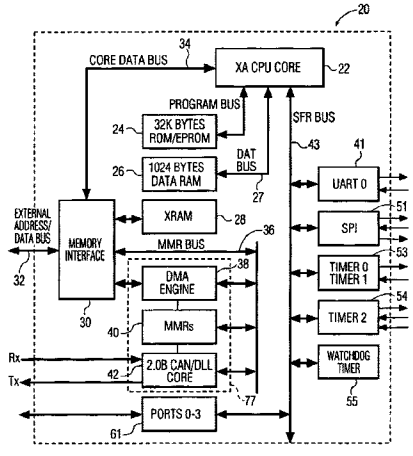
DE 4340219 A1 6/1995 ..... G06F/9/46
GB 2293470 A 3/1996 ..... G06F/9/44
WO WO8400836 3/1984 ..... G06F/13/00

Primary Examiner—Abdelmoniem Elamin

(57) **ABSTRACT**

A CAN microcontroller that supports a plurality of uniquely-numbered message objects, that includes a processor core that runs CAN applications, a plurality of message buffers associated with respective ones of the message objects, and a CAN/CAL module. The CAN microcontroller further includes a plurality of individual message object registers associated with each message object, including at least one control register that contains an interrupt-enable control bit, a receive enable bit, and a transmit enable bit. The CAN microcontroller also includes a plurality of global message object control registers, including at least one message complete status register that contains a plurality of status flag bits for respective ones of the message objects, at least one interrupt flag register that contains a receive complete interrupt flag bit and a transmit complete interrupt flag bit, and a message complete info register that contains a plurality of message object identification bits and a status bit. The CAN/CAL module includes a message handling function that automatically transfers successive frames of an incoming multi-frame message to the message buffer associated with a corresponding message object; an end-of-message detection function that detects an end-of-message condition which occurs when the last frame of the accepted incoming multi-frame message has been stored in the message buffer associated with the corresponding message object; and, an end-of-message detection handling and interrupt generation function that, in response to the detection of the end-of-message condition: sets the status flag bit contained in the at least one message complete status register associated with the corresponding message object; sets the receive complete interrupt flag bit contained in the at least one interrupt flag register, if the interrupt-enable control bit contained in the at least one control register associated with the corresponding message object is set; and, sets the status bit contained in the message complete info register, if the interrupt-enable control bit contained in the at least one control register associated with the corresponding message object is set.

31 Claims, 7 Drawing Sheets



U.S. PATENT DOCUMENTS

4,802,089	A	1/1989	Shaw	364/200
5,349,667	A	9/1994	Kaneko	395/725
5,471,620	A	11/1995	Shimizu et al.	395/375
5,530,597	A	6/1996	Bowles et al.	395/735
5,881,063	A *	3/1999	Bement et al.	370/389
6,363,083	B1 *	3/2002	Spielbauer et al.	370/470
6,529,594	B1 *	3/2003	Brockman et al.	379/133

* cited by examiner

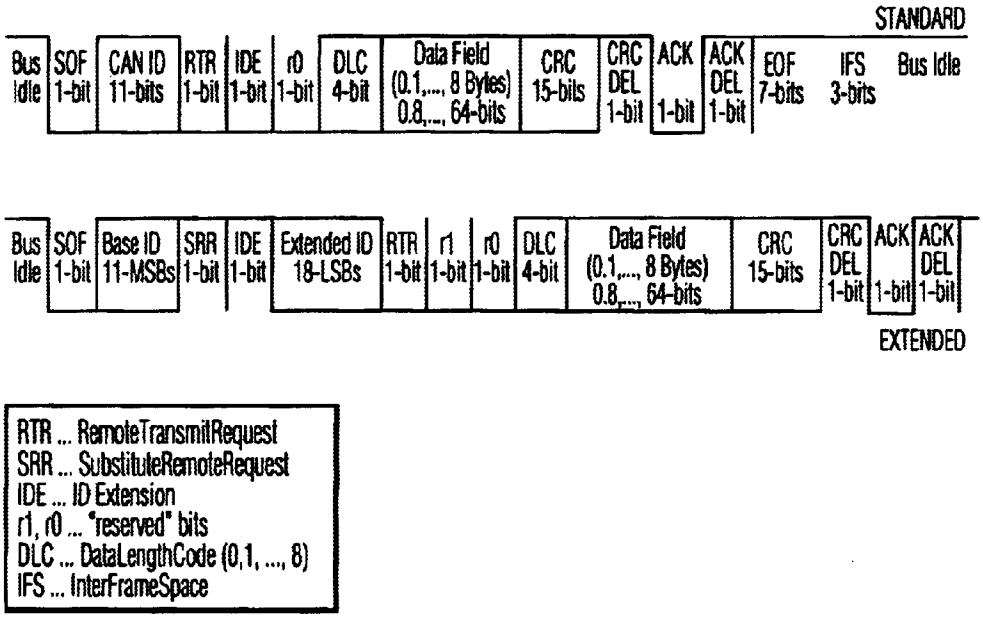

STANDARD
Bus Idle
SOF 1-bit
CAN ID 11-bits
RTR 1-bit
IDE 1-bit
r0 1-bit
DLC 4-bit
Data Field (0.1,..., 8 Bytes) 0.8,..., 64-bits
CRC 15-bits
CRC DEL 1-bit
ACK 1-bit
ACK DEL 1-bit
EOF 7-bits
IFS 3-bits
Bus Idle
Bus Idle
SOF 1-bit
Base ID 11-MSBs
SRR 1-bit
IDE 1-bit
Extended ID 18-LSBs
RTR 1-bit
r1 1-bit
r0 1-bit
DLC 4-bit
Data Field (0.1,..., 8 Bytes) 0.8,..., 64-bits
CRC 15-bits
CRC DEL 1-bit
ACK 1-bit
ACK DEL 1-bit
EXTENDED
RTR ... RemoteTransmitRequest
SRR ... SubstituteRemoteRequest
IDE ... ID Extension
r1, r0 ... "reserved" bits
DLC ... DataLengthCode (0,1, ..., 8)
IFS ... InterFrameSpace

FIG. 1

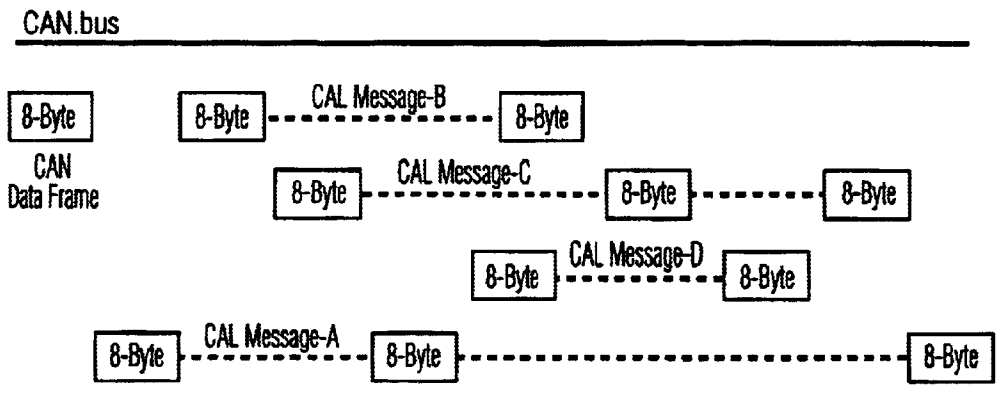

CAN.bus
8-Byte
CAN Data Frame
8-Byte
CAL Message-B
8-Byte
8-Byte
CAL Message-C
8-Byte
8-Byte
8-Byte
CAL Message-D
8-Byte
CAL Message-A
8-Byte
8-Byte

FIG. 2

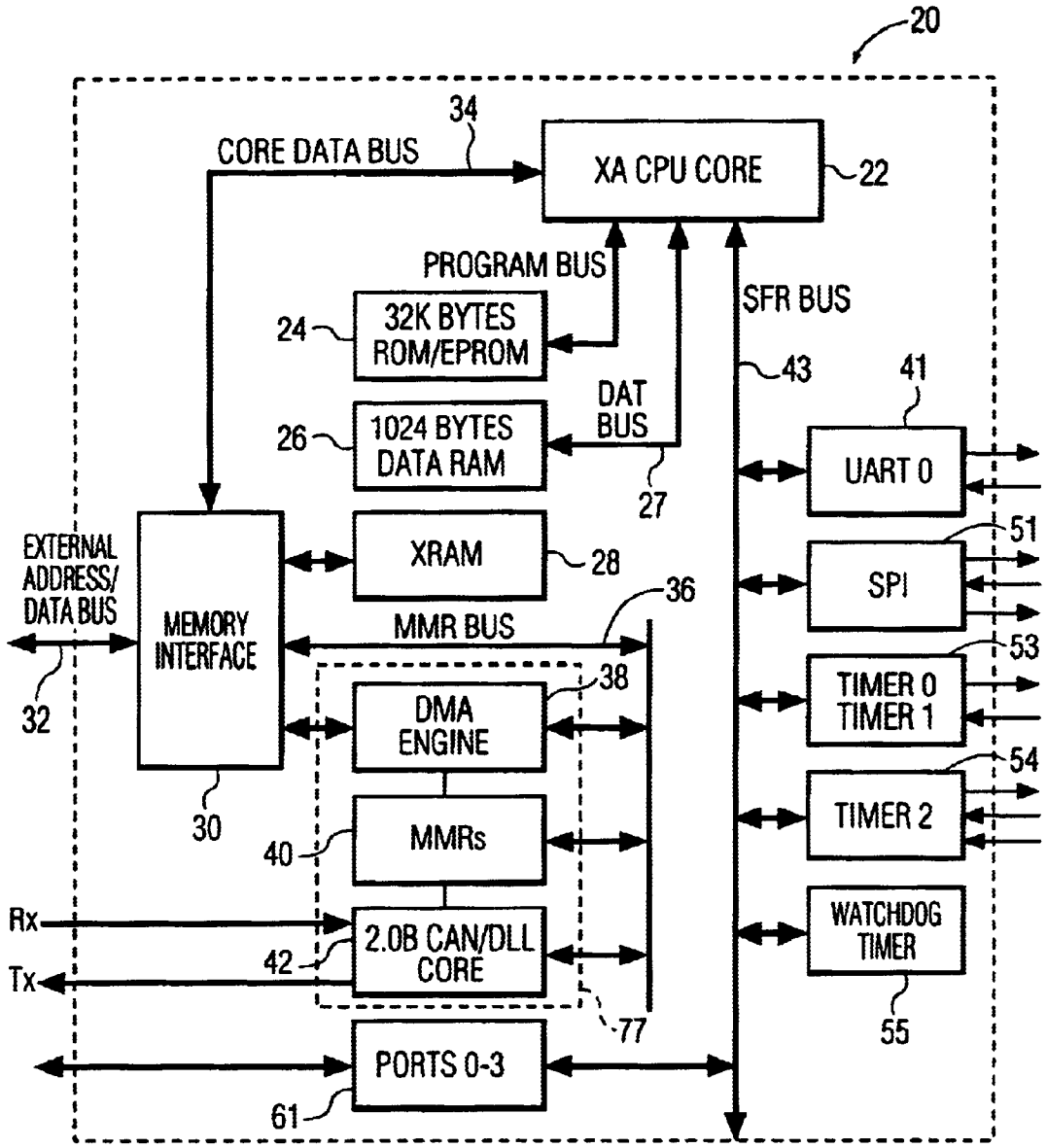

20
34
CORE DATA BUS
XA CPU CORE
22
PROGRAM BUS
SFR BUS
24
32K BYTES ROM/EPROM
43
41
DAT BUS
26
1024 BYTES DATA RAM
27
UART 0
EXTERNAL ADDRESS/ DATA BUS
XRAM
28
51
SPI
36
MMR BUS
MEMORY INTERFACE
53
TIMER 0 TIMER 1
38
DMA ENGINE
32
54
TIMER 2
30
40
MMRs
Rx
WATCHDOG TIMER
2.0B CAN/DLL CORE
42
Tx
77
55
PORTS 0-3
61

FIG. 3

MMRs

MMR name	R/W?	Reset	Access	Address Offset	Description
Message Object Registers (n = 0 - 31)					
MnMIDH	R/W	x....x00b	Word only	000n4n3n2n1n0 0000b (n0h)	Message n Match ID High
MnMIDL	R/W	xxxxh	Word only	000n4n3n2n1n0 0010b (n2h)	Message n Match ID Low
MnMSKH	R/W	x....x000b	Word only	000n4n3n2n1n0 0100b (n4h)	Message n Mask High
MnMSKL	R/W	xxxxh	Word only	000n4n3n2n1n0 0110b (n6h)	Message n Mask Low
MnCTL	R/W	00000xxxb	Byte/Word	000n4n3n2n1n0 1000b (n8h)	Message n Control
MnBLR	R/W	xxxxh	Word only	000n4n3n2n1n0 1010b (nAh)	Message n Buffer Location
MnBSZ	R/W	00000xxxb	Byte/Word	000n4n3n2n1n0 1100b (nCh)	Message n Buffer Size
MnFCR	R/W	00xxxxxxb	Byte/Word	000n4n3n2n1n0 1110b (nEh)	Message n Fragmentation Count
CIC Registers					
MCPLL	R/C	0000h	Byte/Word	224h	Message Complete Low
MCPLH	R/C	0000h	Byte/Word	226h	Message Complete High
CANINTFLG	R/C	0000h	Byte/Word	228h	CAN Interrupt Flag Register
MCIR	RO	0000h	Byte/Word	229h	Message Complete Info Reg.
MEIR	RO	0000h	Byte/Word	22Ah	Message Error Info Register
FESTR	R/C	0000h	Byte/Word	22Ch	Frame Error Status Register
FEENR	R/W	0000h	Byte/Word	22Eh	Frame Error Enable Register
SCP/SPI Registers					
SPICFG	R/W	0000h	Byte/Word	260h	SCP/SPI Configuration
SPIDATA	R/W	00h	Byte/Word	262h	SCP/SPI Data
SPICS	R/W	00h	Byte/Word	263h	SCP/SPI Control and Status
CCB Registers					
CANCMR	R/W	01h	Byte/Word	270h	CAN Command Register
CANSTR	R/O	00h	Byte/Word	271h	CAN Status Register
CANBTR	R/W	00h	Byte/Word	272h	CAN Bus Timing Reg. (low)
-	R/W	00h	Byte/Word	273h	CAN Bus Timing Reg. (high)
TXERC	R/W*	00h	Byte/Word	274h	Tx Error Counter
RXERC	R/W*	00h	Byte/Word	275h	Rx Error Counter
EWLR	R/W	96h	Byte/Word	276h	Error Warning Limit Register
ECCR	RO	0000h	Byte/Word	278h	Error Code Capture Register
ALCR	RO	0000h	Byte/Word	27Ah	Arbitration Lost Capture Reg.
RTXDTM	WO	0000h	Byte/Word	27Ch	RTX Data Test Mode
GCTL	R/W	0000h	Byte/Word	27Eh	Global Control Byte
MIF Registers					
XRAMB	R/W	FEh	Byte/Word	290h	XRAM Base Address
MBXSR	R/W	FFh	Byte/Word	291h	Msg. Buff./XRAM Seg. Reg.
MIFBTRL	R/W	EFh	Byte/Word	292h	MIF Bus Timing Reg. Low
MIFBTRH	R/W	FFh	Byte/Word	293h	MIF Bus Timing Reg. High

Legend: R/W = Read & Write, RO = Read Only, WO = Write Only, R/C = Read & Clear, W* = Writable only during CAN Reset mode, x = undefined after reset.

FIG. 4

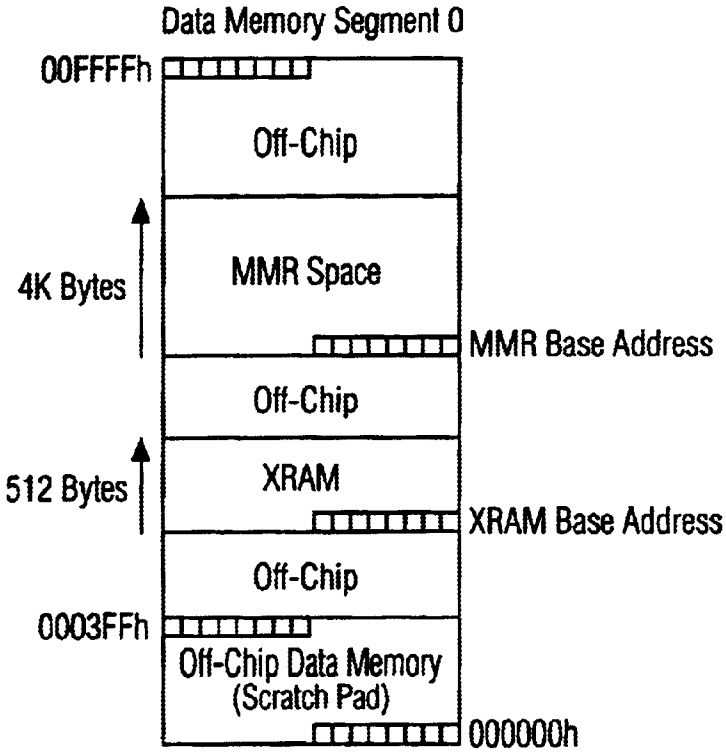

Data Memory Segment 0
00FFFFh
Off-Chip
4K Bytes
MMR Space
MMR Base Address
Off-Chip
512 Bytes
XRAM
XRAM Base Address
Off-Chip
0003FFh
Off-Chip Data Memory
(Scratch Pad)
000000h

FIG. 5

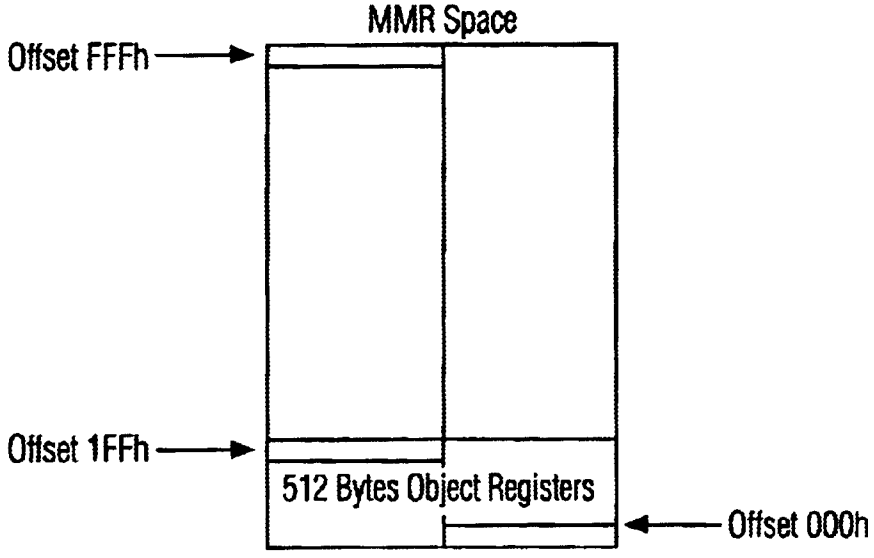

MMR Space
Offset FFFh
Offset 1FFh
512 Bytes Object Registers
Offset 000h

FIG. 6

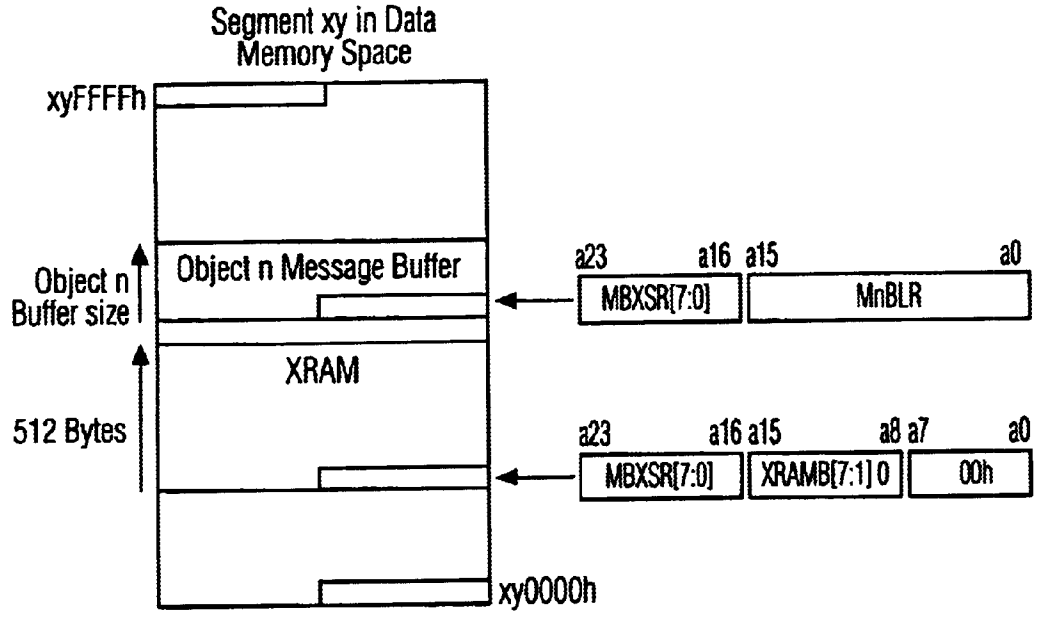

Segment xy in Data Memory Space
xyFFFFh
Object n Message Buffer
Object n Buffer size
XRAM
512 Bytes
xy0000h
a23
a16
a15
a0
MBXSR[7:0]
MnBLR
a23
a16
a15
a8
a7
a0
MBXSR[7:0]
XRAMB[7:1] 0
00h

FIG. 7

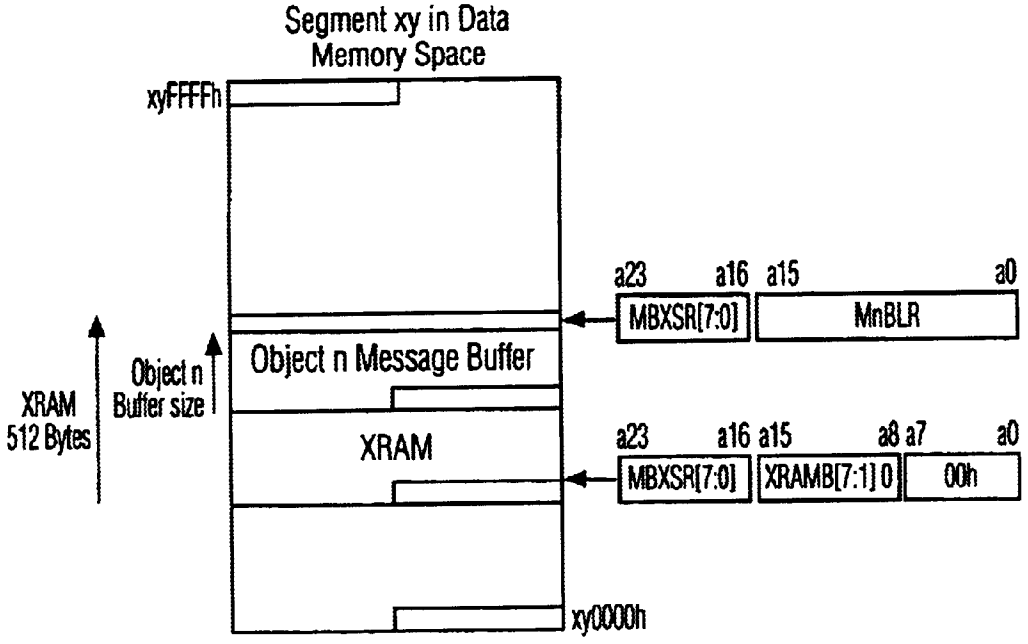

Segment xy in Data Memory Space
xyFFFFh
Object n Message Buffer
Object n Buffer size
XRAM
512 Bytes
XRAM
xy0000h
a23
a16
a15
a0
MBXSR[7:0]
MnBLR
a23
a16
a15
a8
a7
a0
MBXSR[7:0]
XRAMB[7:1] 0
00h

FIG. 8

Object n Match ID Field (MnMIDH and MnMIDL)

Mid28 – Mid18	Mid17 – Mid10	Mid9 – Mid2	Mid1	Mid0	MIDE

Object n Mask Field (MnMSKH and MnMSKL)

Msk28 – Msk18	Msk17 – Msk10	Msk9 – Msk2	Msk1	Msk0

Screener ID Field (assembled from incoming bit-stream)

CAN ID.28 – CAN ID.18	Data Byte 1 [7:0]	Data Byte 2 [7:0]	x	x	IDE

FIG. 9

Object n Match ID Field (MnMIDH and MnMIDL)

Mid28 – Mid18	Mid17 – Mid10	Mid9 – Mid2	Mid1	Mid0	MIDE

Object n Mask Field (MnMSKH and MnMSKL)

Msk28 – Msk18	Msk17 – Msk10	Msk9 – Msk2	Msk1	Msk0

Screener ID Field (assembled from incoming bit-stream)

CAN ID.28 – CAN ID.0	IDE

FIG. 10

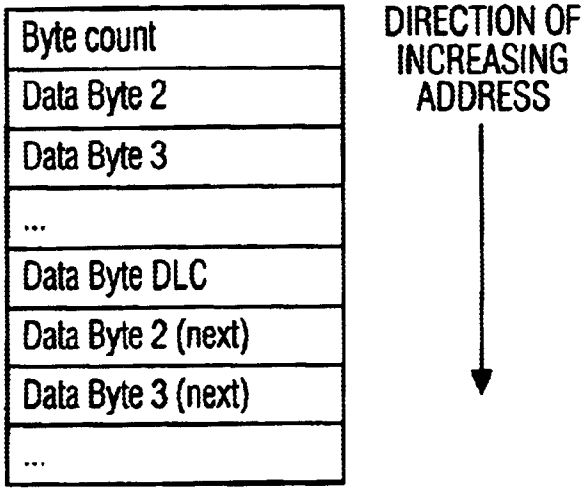

Byte count
Data Byte 2
Data Byte 3
...
Data Byte DLC
Data Byte 2 (next)
Data Byte 3 (next)
...
DIRECTION OF INCREASING ADDRESS

FIG. 11

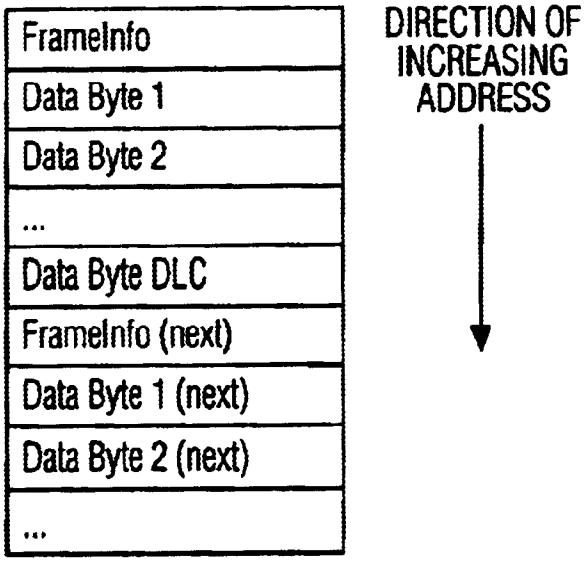

FrameInfo
Data Byte 1
Data Byte 2
...
Data Byte DLC
FrameInfo (next)
Data Byte 1 (next)
Data Byte 2 (next)
...
DIRECTION OF INCREASING ADDRESS

FIG. 12

END-OF-MESSAGE HANDLING AND INTERRUPT GENERATION IN A CAN MODULE PROVIDING HARDWARE ASSEMBLY OF MULTI-FRAME CAN MESSAGES

This application claims the full benefit and priority of U.S. Provisional Application Serial No. 60/154,022, filed on Sep. 15, 1999, the disclosure of which is fully incorporated herein for all purposes.

BACKGROUND OF THE INVENTION

The present invention relates generally to the field of data communications, and more particularly, to the field of serial communications bus controllers and microcontrollers that incorporate the same.

CAN (Control Area Network) is an industry-standard, two-wire serial communications bus that is widely used in automotive and industrial control applications, as well as in medical devices, avionics, office automation equipment, consumer appliances, and many other products and applications. CAN controllers are currently available either as stand-alone devices adapted to interface with a microcontroller or as circuitry integrated into or modules embedded in a microcontroller chip. Since 1986, CAN users (software programmers) have developed numerous high-level CAN Application Layers (CALs) which extend the capabilities of the CAN while employing the CAN physical layer and the CAN frame format, and adhering to the CAN specification. CALs have heretofore been implemented primarily in software, with very little hardware CAL support. Consequently, CALs have heretofore required a great deal of host CPU intervention, thereby increasing the processing overhead and diminishing the performance of the host CPU.

Thus, there is a need in the art for a CAN hardware implementation of CAL functions normally implemented in software in order to offload these tasks from the host CPU to the CAN hardware, thereby enabling a great savings in host CPU processing resources and a commensurate improvement in host CPU performance. One of the most demanding and CPU resource-intensive CAL functions is message management, which entails the handling, storage, and processing of incoming CAL/CAN messages received over the CAN serial communications bus and/or outgoing CAL/CAN messages transmitted over the CAN serial communications bus. CAL protocols, such as DeviceNet, CANopen, and OSEK, deliver long messages distributed over many CAN frames, which methodology is sometimes referred to as "fragmented" or "segmented" messaging. The process of assembling such fragmented, multi-frame messages has heretofore required a great deal of host CPU intervention. In particular, CAL software running on the host CPU actively monitors and manages the buffering and processing of the message data, in order to facilitate the assembly of the message fragments or segments into complete messages.

Based on the above and foregoing, it can be appreciated that there presently exists a need in the art for a hardware implementation of CAL functions normally implemented in software in order to offload these tasks from the host CPU, thereby enabling a great savings in host CPU processing resources and a commensurate improvement in host CPU performance.

The assignee of the present invention has recently developed a new microcontroller product, designated "XA-C3", that fulfills this need in the art. The XA-C3 is the newest

member of the Philips XA (eXtended Architecture) family of high performance 16-bit single-chip microcontrollers. It is believed that the XA-C3 is the first chip that features hardware CAL support.

The XA-C3 is a CMOS 16-bit CAL/CAN 2.0B microcontroller that incorporates a number of different inventions, including the present invention. These inventions include novel techniques and hardware for filtering, buffering, handling, and processing CAL/CAN messages, including the automatic assembly of multi-frame fragmented messages with minimal CPU intervention, as well as for managing the storage and retrieval of the message data, and the memory resources utilized therefor. In particular, the XA-C3 CAN module has the unique ability to track and reassemble the packets constituting a fragmented message, completely in hardware, only interrupting the CPU (processor core) once a complete, multi-frame message is received and assembled. This tremendously reduces the processor bandwidth required for message handling, thereby significantly increasing available bandwidth for other tasks, so that system performance is greatly enhanced.

The present invention relates to the techniques employed by the XA-C3 microcontroller for detecting an end-of-message condition, for end-of-message handling, and for generating the appropriate end-of-message interrupt. Fundamentally, the task of responding to the end of a message should be very straightforward. More particularly, the final frame of the message should be stored in the buffer, an interrupt to the processor should be generated, and the software should respond by retrieving the message data from the buffer.

However, this seemingly fundamental task is greatly complicated in the XA-C3 microcontroller, since the XA-C3 CAN module can concurrently assemble many (up to 32) incoming, fragmented messages of varying lengths, whereby up to 32 completed messages can be staged and waiting by the time the processor responds to the initial end-of-message interrupt, i.e., the interrupt issued in response to completion of the first received complete message. A further complication arises by virtue of the fact that it is often appropriate for the software to "poll" certain categories of messages on an occasional basis rather than respond to an end-of-message interrupt at the moment messages within one of these categories completes. This implies that some message objects may be set up to generate an end-of-message interrupt, while others are not. Either way, the software must be able to determine at any time whether a complete message is available for all message objects. Further, when an end-of-message interrupt is asserted, the processor must be able to determine quickly and easily which message or messages are complete, i.e., ready for processing.

In designing the XA-C3 microcontroller, the present inventors contemplated and rejected a number of message-complete handling schemes, because these schemes would have required extremely cumbersome, inefficient software code, and/or would have added far too much die area. The present invention, as described below, was conceived and finally adopted as the optimum approach.

SUMMARY OF THE INVENTION

The present invention encompasses a CAN microcontroller that supports a plurality of uniquely-numbered message objects, that includes a processor core that runs CAN applications, a plurality of message buffers associated with respective ones of the message objects, and a CAN/CAL

module. The CAN microcontroller further includes a plurality of individual message object registers associated with each message object, including at least one control register that contains an interrupt-enable control bit, a receive enable bit, and a transmit enable bit. The CAN microcontroller also includes a plurality of global message object control registers, including at least one message complete status register that contains a plurality of status flag bits for respective ones of the message objects, at least one interrupt flag register that contains a receive complete interrupt flag bit and a transmit complete interrupt flag bit, and a message complete info register that contains a plurality of message object identification bits and a status bit.

The CAN/CAL module includes an acceptance filtering function that performs acceptance filtering on each incoming, multi-frame message by comparing a screener field of the incoming, multi-frame message with an acceptance filter field associated with each message object which has its associated receive enable bit set, wherein the incoming, multi-frame message is accepted if its screener field matches the acceptance filter field of a receive-enabled message object; a message handling function that automatically transfers successive frames of an accepted incoming multi-frame message to the message buffer associated with the matching receive-enabled message object; an end-of-message detection function that detects an end-of-message condition which occurs when the last flame of the accepted incoming multi-frame message has been stored in the message buffer associated with the matching receive-enabled message object; and, an end-of-message detection handling and interrupt generation function that, in response to the detection of the end-of-message condition: sets the status flag bit contained in the at least one message complete status register corresponding to the matching receive-enabled message object; sets the receive complete interrupt flag bit contained in the at least one interrupt flag register, if the interrupt-enable control bit contained in the at least one control register associated with the matching receive-enabled message object is set; and, sets the status bit contained in the message complete info register, if the interrupt-enable control bit contained in the at least one control register associated with the matching receive-enabled message object is set.

A current application running on the processor core can check the status of the status flag bits contained in the at least one message complete status register, at selected times. The current application running on the processor core processes the completed message corresponding to the message object associated with an enabled status flag bit that is contained in the at least one message complete status register.

A current application running on the processor core can also check the status of the status bit contained in the message complete info register to determine whether or not there are any pending completed messages associated with a respective interrupt-enabled message object. In response to a determination that there is a pending completed message based on the status of the status bit contained in the message complete info register, the current application running on the processor core: processes the completed message corresponding to the lowest-numbered receive-enabled message object identified by the message object identification bits contained in the message complete info register; clears the status flag bit contained in the at least one control register associated with the lowest-numbered receive-enabled message object; checks the status of the status bit contained in the message complete info register; and repeats each of the above-recited operations if the status bit contained in the message complete info register is enabled, until the status flag bit is no longer enabled.

The CAN/CAL module generates a message-complete interrupt in response to detection of an end-of-message condition if the interrupt-enable control bit contained in the at least one control register associated with the corresponding receive-enabled message object is enabled. The current application running on the processor core processes the completed message, in response to the message-complete interrupt.

BRIEF DESCRIPTION OF THE DRAWINGS

These and various other aspects, features, and advantages of the present invention will be readily understood with reference to the following detailed description of the invention read in conjunction with the accompanying drawings, in which:

FIG. **1** is a diagram illustrating the format of a Standard CAN Frame and the format of an Extended CAN Frame;

FIG. **2** is a diagram illustrating the interleaving of CAN Data Frames of different, unrelated messages;

FIG. **3** is a high-level, functional block diagram of the XA-C3 microcontroller;

FIG. **4** is a table listing all of the Memory Mapped Registers (MMRs) provided by the XA-C3 microcontroller;

FIG. **5** is a diagram illustrating the mapping of the overall data memory space of the XA-C3 microcontroller;

FIG. **6** is a diagram illustrating the MMR space contained within the overall data memory space of the XA-C3 microcontroller;

FIG. **7** is a diagram illustrating formation of the base address of the on-chip XRAM of the XA-C3 microcontroller, with an object n message buffer mapped into off-chip data memory;

FIG. **8** is a diagram illustrating formation of the base address of the on-chip XRAM of the XA-C3 microcontroller, with an object n message buffer mapped into the on-chip XRAM;

FIG. **9** is a diagram illustrating the Screener ID Field for a Standard CAN Frame;

FIG. **10** is a diagram illustrating the Screener ID Field for an Extended CAN Frame;

FIG. **11** is a diagram illustrating the message storage format for fragmented CAL messages; and,

FIG. **12** is a diagram illustrating the message storage format for fragmented CAN messages.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention is described below in the context of a particular implementation thereof, i.e., in the context of the XA-C3 microcontroller manufactured by Philips Semiconductors. Of course, it should be clearly understood that the present invention is not limited to this particular implementation, as any one or more of the various aspects and features of the present invention disclosed herein can be utilized either individually or any combination thereof, and in any desired application, e.g., in a stand-alone CAN controller device or as part of any other microcontroller or system.

The following terms used herein in the context of describing the preferred embodiment of the present invention (i.e., the XA-C3 microcontroller) are defined as follows:

Standard CAN Frame: The format of a Standard CAN Frame is depicted in FIG. **1**.

Extended CAN Frame: The format of an Extended CAN Frame is also depicted in FIG. **1**.

Acceptance Filtering: The process a CAN device implements in order to determine if a CAN frame should be accepted or ignored and, if accepted, to store that frame in a pre-assigned Message Object.

Message Object: A Receive RAM buffer of pre-specified size (up to 256 bytes for CAL messages) and associated with a particular Acceptance Filter or, a Transmit RAM buffer which the User preloads with all necessary data to transmit a complete CAN Data Frame. A Message Object can be considered to be a communication channel over which a complete message, or a succession of messages, can be transmitted.

CAN Arbitration ID: An 11-bit (Standard CAN 2.0 Frame) or 29-bit (Extended CAN 2.0B Frame) identifier field placed in the CAN Frame Header. This ID field is used to arbitrate Frame access to the CAN bus. Also used in Acceptance Filtering for CAN Frame reception and Transmit Pre-Arbitration.

Screener ID: A 30-bit field extracted from the incoming message which is then used in Acceptance Filtering. The Screener ID includes the CAN Arbitration ID and the IDE bit, and can include up to 2 Data Bytes. These 30 extracted bits are the information qualified by Acceptance Filtering.

Match ID: A 30-bit field pre-specified by the user to which the incoming Screener ID is compared. Individual Match IDs for each of 32 Message Objects are programmed by the user into designated Memory Mapped Registers (MMRs).

Mask: A 29-bit field pre-specified by the user which can override (Mask) a Match ID comparison at any particular bit (or, combination of bits) in an Acceptance Filter. Individual Masks, one for each Message Object, are programmed by the user in designated MMRs. Individual Mask patterns assure that single Receive Objects can Screen for multiple acknowledged CAL/CAN Frames and thus minimize the number of Receive Objects that must be dedicated to such lower priority Frames. This ability to Mask individual Message Objects is an important new CAL feature.

CAL: CAN Application Layer. A generic term for any high-level protocol which extends the capabilities of CAN while employing the CAN physical layer and the CAN frame format, and which adheres to the CAN specification. Among other things, CALs permit transmission of Messages which exceed the 8 byte data limit inherent to CAN Frames. This is accomplished by dividing each message into multiple packets, with each packet being transmitted as a single CAN Frame consisting of a maximum of 8 data bytes. Such messages are commonly referred to as "segmented" or "fragmented" messages. The individual CAN Frames constituting a complete fragmented message are not typically transmitted in a contiguous fashion, but rather, the individual CAN Frames of different, unrelated messages are interleaved on the CAN bus, as is illustrated in FIG. **2**

Fragmented Message: A lengthy message (in excess of 8 bytes) divided into data packets and transmitted using a sequence of individual CAN Frames. The specific ways that sequences of CAN Frames construct these lengthy messages is defined within the context of a specific CAL. The XA-C3 microcontroller automatically re-assembles these packets into the original, lengthy message in hardware and reports (via an interrupt) when the completed (re-assembled) message is available as an associated Receive Message Object.

Message Buffer: A block of locations in XA Data memory where incoming (received) messages are stored or where outgoing (transmit) messages are staged.

MMR: Memory Mapped Register. An on-chip command/control/status register whose address is mapped into XA Data memory space and is accessed as Data memory by the XA processor. With the XA-C3 microcontroller, a set of eight dedicated MMRs are associated with each Message Object. Additionally, there are several MMRs whose bits control global parameters that apply to all Message Objects.

With reference now to FIG. **3**, there can be seen a high-level block diagram of the XA-C3 microcontroller **20**. The XA-C3 microcontroller **20** includes the following functional blocks that are fabricated on a single integrated circuit (IC) chip packaged in a 44-pin PLCC or a 44-pin LQFP package:

- an XA CPU Core **22**, that is currently implemented as a 16-bit fully static CPU with 24-bit program and data address range, that is upwardly compatible with the 80C51 architecture, and that has an operating frequency of up to 30 MHz;
- a program or code memory **24** that is currently implemented as a 32K ROM/EPROM, and that is bi-directionally coupled to the XA CPU Core **22** via an internal Program bus **25**. A map of the code memory space is depicted in FIG. **4**;
- a Data RAM **26** (internal or scratch pad data memory) that is currently implemented as a 1024 Byte portion of the overall XA-C3 data memory space, and that is bi-directionally coupled to the XA CPU Core **22** via an internal DATA bus **27**;
- an on-chip message buffer RAM or XRAM **28** that is currently implemented as a 512 Byte portion of the overall XA-C3 data memory space which may contain part or all of the CAN/CAL (Transmit & Receive Object) message buffers;
- a Memory Interface (MIF) unit **30** that provides interfaces to generic memory devices such as SRAM, DRAM, flash, ROM, and EPROM memory devices via an external address/data bus **32**, via an internal Core Data bus **34**, and via an internal MMR bus **36**;
- a DMA engine **38** that provides **32** CAL DMA Channels;
- a plurality of on-chip Memory Mapped Registers (MMRs) **40** that are mapped to the overall XA-C3 data memory space—a 4K Byte portion of the overall XA-C3 data memory space is reserved for MMRs. These MMRs include 32 (Message) Object or Address Pointers and 32 ID Screeners or Match IDs, corresponding to the 32 CAL Message Objects. A complete listing of all MMRs is provided in the Table depicted in FIG. **5**;
- a 2.0B CAN/DLL Core **42** that is the CAN Controller Core from the Philips SJA1000 CAN (2.0A/B) Data Link Layer (CDLL) device (hereinafter referred to as the "CAN Core Block" (CCB)); and,
- an array of standard microcontroller peripherals that are bi-directionally coupled to the XA CPU Core **22** via a Special Function Register (SFR) bus **43**. These standard microcontroller peripherals include Universal Asynchronous Receiver Transmitter (UART) **49**, an SPI serial interface (port) **51**, three standard timers/counters with toggle output capability, namely, Timer **0** & Timer **1** included in Timer block **53**, and Timer **2** included in Timer block **54**, a Watchdog Timer **55**, and four 8-bit I/O ports, namely, Ports **0–3** included in

block **61**, each of which has 4 programmable output configurations.

The DMA engine **38**, the MMRs **40**, and the CCB **42** can collectively be considered to constitute a CAN/CAL module **77**, and will be referred to as such at various times throughout the following description. Further, the particular logic elements within the CAN/CAL module **77** that perform "message management" and "message handling" functions will sometimes be referred to as the "message management engine" and the "message handler", respectively, at various times throughout the following description. Other nomenclature will be defined as it introduced throughout the following description.

As previously mentioned, the XA-C3 microcontroller **20** automatically implements, in hardware, many message management and other functions that were previously only implemented in software running on the host CPU (or not implemented at all), including transparent, automatic re-assembly of up to 32 concurrent, interleaved, multi-frame, fragmented CAL messages. For each application that is installed to run on the host CPU (i.e., the XA CPU Core **22**), the user (software programmer) must set-up the hardware for performing these functions by programming certain ones of the MMRs and SFRs in the manner set forth in the XA-C3 Functional Specification and XA-C3 CAN Transport Layer Controller User Manual. The register programming procedures that are most relevant to an understanding of the present invention are described below, followed by a description of the various message management and other functions that are automatically performed by the CAL/CAN module **77** during operation of the XA-C3 microcontroller **20** after it has been properly set-up by the user. Following these sections, a more detailed description of the particular invention to which this application is directed is provided.

Set-up/Programming Procedures

As an initial matter, the user must map the overall XA-C3 data memory space, as illustrated in FIG. **5**. In particular, subject to certain constraints, the user must specify the starting or base address of the XRAM **28** and the starting or base address of the MMRs **40**. The base address of the MMRs **40** can be specified by appropriately programming Special Function Registers (SFRs) MRBL and MRBH. The base address of the XRAM **28** can be specified by appropriately programming the MMRs designated MBXSR and XRAMB (see FIG. **4**).

The user can place the 4K Byte space reserved for MMRs **40** anywhere within the entire 16 Mbyte data memory space supported by the XA architecture, other than at the very bottom of the memory space (i.e., the first 1K Byte portion, starting address of 000000h), where it would conflict with the on-chip Data RAM **26** that serves as the internal or scratch-pad memory. The 4K Bytes of MMR space will always start at a 4K boundary. The reset values for MRBH and MRBL are 0Fh and F0h, respectively. Therefore, after a reset, the MMR space is mapped to the uppermost 4K Bytes of Data Segment 0Fh, but access to the MMRs **40** is disabled. The first 512 Bytes (offset 000h–1FFh) of MMR space are the Message Object Registers (eight per Message Object) for objects n=0–31, as is shown in FIG. **6**.

The base address of the XRAM **28** is determined by the contents of the MMRs designated MBXSR and XRAMB, as is shown in FIGS. **7** and **8**. As previously mentioned, the 512 Byte XRAM **28** is where some (or all) of the 32 (Rx/Tx) message buffers (corresponding to Message Objects n=0–31) reside. The message buffers can be extended off-chip to a maximum of 8 K Bytes. This off-chip expansion capability can accommodate up to thirty-two, 256-Byte message buffers. Since the uppermost 8 bits of all message buffer addresses are formed by the contents of the MBXSR register, the XRAM **28** and all 32 message buffers must reside in the same 64K Byte data memory segment. Since the XA-C3 microcontroller **20** only provides address lines **A0–A19** for accessing external memory, all external memory addresses must be within the lowest 1MByte of address space. Therefore, if there is external memory in the system into which any of the 32 message buffers will be mapped, then all 32 message buffers and the XRAM **28** must also be mapped entirely into that same 64K Byte segment, which must be below the 1MByte address limit.

After the memory space has been mapped, the user can set-up or define up to 32 separate Message Objects, each of which can be either a Transmit (Tx) or a Receive (Rx) Message Object. A Rx Message Object can be associated either with a unique CAN ID, or with a set of CAN IDs which share certain ID bit fields. As previously mentioned, each Message Object has its own reserved block of data memory space (up to 256 Bytes), which is referred to as that Message Object's message buffer. As will be seen, both the size and the base address of each Message Object's message buffer is programmable.

As previously mentioned, each Message Object is associated with a set of eight MMRs **40** dedicated to that Message Object. Some of these registers function differently for Tx Message Objects than they do for Rx Message Objects. These eight MMRs **40** are designated "Message Object Registers" (see FIG. **4**). The names of these eight MMRs **40** are:

1.	MnMIDH	Message n Match ID High
2.	MnMIDL	Message n Match ID Low
3.	MnMSKH	Message n Mask High
4.	MnMSKL	Message n Mask Low
5.	MnCTL	Message n Control
6.	MnBLR	Message n Buffer Location Register
7.	MnBSZ	Message n Buffer Size
8.	MnFCR	Message n Fragment Count Register

where n ranges from 0 to 31 (i.e., corresponding to 32 independent Message Objects).

In general, the user defines or sets up a Message Object by configuring (programming) some or all of the eight MMRs dedicated to that Message Object, as will be described below. Additionally, as will be described below, the user must configure (program) the global GCTL register, whose bits control global parameters that apply to all Message Objects.

In particular, the user can specify the Match ID value for each Message Object to be compared against the Screener IDs extracted from incoming CAN Frames for Acceptance Filtering. The Match ID value for each Message Object n is specified in the MnMIDH and MnMIDL registers associated with that Message Object n. The user can mask any Screener ID bits which are not intended to be used in Acceptance Filtering, on an object-by-object basis, by writing a logic '1' in the desired (to-be-masked) bit position(s) in the appropriate MnMSKH and/or MnMSKL registers associated with each particular Message Object n. The user is responsible, on set-up, for assigning a unique message buffer location for each Message Object n. In particular, the user can specify the least significant 16 bits of the base address of the message buffer for each particular Message Object n by programming

the MnBLR register associated with that Message Object n. The upper 8 bits of the 24-bit address, for all Message Objects, are specified by the contents of the MBXSR register, as previously discussed, so that the message buffers for all Message Objects reside within the same 64 K Byte memory segment. The user is also responsible, on set-up, for specifying the size of the message buffer for each Message Object n. In particular, the user can specify the size of the message buffer for each particular Message Object n by programming the MnBSZ register associated with that Message Object n. The top location of the message buffer for each Message Object n is determined by the size of that message buffer as specified in the corresponding MnBSZ register.

The user can configure (program) the MnCTL register associated with each particular Message Object n in order to enable or disable that Message Object n, in order to define or designate that Message Object n as a Tx or Rx Message Object; in order to enable or disable automatic hardware assembly of fragmented Rx messages (i.e., automatic fragmented message handling) for that Message Object n; in order to enable or disable automatic generation of a Message-Complete Interrupt for that Message Object n; and, in order to enable or not enable that Message Object n for Remote Transmit Request (RTR) handling. In CANopen and OSEK systems, the user must also initialize the MnFCR register associated with each Message Object n.

As previously mentioned, on set-up, the user must configure (program) the global GCTL register, whose bits control global parameters that apply to all Message Objects. In particular, the user can configure (program) the GCTL register in order to specify the high-level CAL protocol (if any) being used (e.g., DeviceNet, CANopen, or OSEK); in order to enable or disable automatic acknowledgment of CANopen Frames (CANopen auto-acknowledge); and, in order to specify which of two transmit (Tx) pre-arbitration schemes/policies is to be utilized (i.e., either Tx pre-arbitration based on CAN ID, with the object number being used as a secondary tie-breaker, or Tx pre-arbitration based on object number only).

Receive Message Objects and the Receive Process

During reception (i.e., when an incoming CAN Frame is being received by the XA-C3 microcontroller **20**), the CAN/CAL module **77** will store the incoming CAN Frame in a temporary (13-Byte) buffer, and determine whether a complete, error-free CAN frame has been successfully received. If it is determined that a complete, error-free CAN Frame has been successfully received, then the CAN/CAL module **77** will initiate Acceptance Filtering in order to determine whether to accept and store that CAN Frame, or to ignore/discard that CAN Frame.

Acceptance Filtering

In general, because the XA-C3 microcontroller **20** provides the user with the ability to program separate Match ID and Mask fields for each of the 32 independent Message Objects, on an object-by-object basis, as described previously, the Acceptance Filtering process performed by the XA-C3 microcontroller **20** can be characterized as a "match and mask" technique. The basic objective of this Acceptance Filtering process is to determine whether a Screener ID field of the received CAN Frame (excluding the "don't care" bits masked by the Mask field for each Message Object) matches the Match ID of any enabled one of the 32 Message Objects that has been designated a Receive Message Object. If there is a match between the received CAN Frame and more than one Message Object, then the received CAN Frame will be deemed to have matched the Message Object with the lowest object number (n).

Message Storage:

Each incoming (received) CAN Frame that passes Acceptance Filtering, will be automatically stored, via the DMA engine **38**, into the message buffer for the Receive Message Object that particular CAN Frame was found to have matched. In an exemplary implementation, the message buffers for all Message Objects are contained in the XRAM **28**.

Message Assembly:

In general, the DMA engine **38** will transfer each accepted CAN Frame from the 13-byte pre-buffer to the appropriate message buffer (e.g., in the XRAM **28**), one word at a time, starting from the address pointed to by the contents of the MBXSR and MnBLR registers. Every time the DMA engine **38** transfers a byte or a word, it has to request the bus. In this regard, the MIF unit **30** arbitrates between accesses from the XA CPU Core **22** and from the DMA engine **38**. In general, bus arbitration is done on an "alternate" policy. After a DMA bus access, the XA CPU Core **22** will be granted bus access, if requested. After an XA CPU bus access, the DMA engine **38** will be granted bus access, if requested. (However, a burst access by the XA CPU Core **22** cannot be interrupted by a DMA bus access).

Once bus access is granted by the MIF unit **30**, the DMA engine **38** will write data from the 13-byte pre-buffer to the appropriate message buffer location. The DMA engine **38** will keep requesting the bus, writing message data sequentially to the appropriate message buffer location until the whole accepted CAN Frame is transferred. After the DMA engine **38** has successfully transferred an accepted CAN Frame to the appropriate message buffer location, the contents of the message buffer will depend upon whether the message that the CAN Frame belongs to is a non-fragmented (single frame) message or a fragmented message. Each case is described below:

Non-Fragmented Message Assembly:

For Message Objects that have been set up with automatic fragmented message handling disabled (not enabled—i.e., the FRAG bit in the MnCTL register for that Message Object is set to '0'), the complete CAN ID of the accepted CAN Frame (which is either 11 or 29 bits, depending on whether the accepted CAN Frame is a Standard or Extended CAN Frame) is written into the MnMIDH and MnMIDL registers associated with the Message Object that has been deemed to constitute a match, once the DMA engine **38** has successfully transferred the accepted CAN Frame to the message buffer associated with that Message Object. This will permit the user application to see the exact CAN ID which resulted in the match, even if a portion of the CAN ID was masked for Acceptance Filtering. As a result of this mechanism, the contents of the MnMIDH and MnMIDL registers can change every time an incoming CAN Frame is accepted. Since the incoming CAN Frame must pass through the Acceptance Filter before it can be accepted, only the bits that are masked out will change. Therefore, the criteria for match and mask Acceptance Filtering will not change as a result of the contents of the MnMIDH and MnMIDL registers being changed in response to an accepted incoming CAN Frame being transferred to the appropriate message buffer.

Fragmented Message Assembly:

For Message Objects that have been set up with automatic fragmented message handling enabled (i.e., with the FRAG bit in the MnCTL register for that Message Object set to '1'), masking of the 11/29 bit CAN ID field is disallowed. As

such, the CAN ID of the accepted CAN Frame is known unambiguously, and is contained in the MnMIDH and MnMIDL registers associated with the Message Object that has been deemed to constitute a match. Therefore, there is no need to write the CAN ID of the accepted CAN Frame into the MnMIDH and MnMIDL registers associated with the Message Object that has been deemed to constitute a match.

As subsequent CAN Frames of a fragmented message are received, the new data bytes are appended to the end of the previously received and stored data bytes. This process continues until a complete multi-frame message has been received and stored in the appropriate message buffer.

Under CAL protocols DeviceNet, CANopen, and OSEK, if a Message Object is an enabled Receive Message Object, and its associated MnCTL register has its FRAG bit set to '1' (i.e., automatic fragmented message assembly is enabled for that particular Receive Message Object), then the first data byte (Data Byte **1**) of each received CAN Frame that matches that particular Receive Message Object will be used to encode fragmentation information only, and thus, will not be stored in the message buffer for that particular Receive Message Object. Thus, message storage for such "FRAG-enabled" Receive Message Objects will start with the second data byte (Data Byte **2**) and proceed in the previously-described manner until a complete multi-frame message has been received and stored in the appropriate message buffer. This message storage format is illustrated in FIG. **11**. The message handler hardware will use the fragmentation information contained in Data Byte **1** of each CAN Frame to facilitate this process.

Under the CAN protocol, if a Message Object is an enabled Receive Message Object, and its associated MnCTL register has its FRAG bit set to '1' (i.e., automatic fragmented message assembly is enabled for that particular Receive Message Object), then the CAN Frames that match that particular Receive Message Object will be stored sequentially in the message buffer for that particular Receive Message Object using the format shown in FIG. **12**.

When writing message data into a message buffer associated with a Message Object n, the DMA engine **38** will generate addresses automatically starting from the base address of that message buffer (as specified in the MnBLR register associated with that Message Object n). Since the size of that message buffer is specified in the MnBSZ register associated with that Message Object n, the DMA engine **38** can determine when it has reached the top location of that message buffer. If the DMA engine **38** determines that it has reached the top location of that message buffer, and that the message being written into that message buffer has not been completely transferred yet, the DMA engine **38** will wrap around by generating addresses starting from the base address of that message buffer again. Some time before this happens, a warning interrupt will be generated so that the user application can take the necessary action to prevent data loss.

The message handler will keep track of the current address location of the message buffer being written to by the DMA engine **38**, and the number of bytes of each CAL message as it is being assembled in the designated message buffer. After an "End of Message" for a CAL message is decoded, the message handler will finish moving the complete CAL message and the Byte Count into the designated message buffer via the DMA engine **38**, and then generate an interrupt to the XA CPU Core **22** indicating that a complete message has been received.

Since Data Byte **1** of each CAN Frame contains the fragmentation information, it will never be stored in the designated message buffer for that CAN Frame. Thus, up to seven data bytes of each CAN Frame will be stored. After the entire message has been stored, the designated message buffer will contain all of the actual informational data bytes received (exclusive of fragmentation information bytes) plus the Byte Count at location **00** which will contain the total number of informational data bytes stored.

It is noted that there are several specific user set-up/programming procedures that must be followed when invoking automatic hardware assembly of fragmented OSEK and CANopen messages. These and other particulars can be found in the XA-C3 CAN Transport Layer Controller User Manual that is part of the parent Provisional Application Serial No. 60/154,022, the disclosure of which has been fully incorporated herein for all purposes.

Transmit Message Objects and the Transmit Process

In order to transmit a message, the XA application program must first assemble the complete message and store it in the designated message buffer for the appropriate Transmit Message Object n. The message header (CAN ID and Frame Information) must be written into the MnMIDH, MnMIDL, and MnMSKH registers associated with that Transmit Message Object n. After these steps are completed, the XA application is ready to transmit the message. To initiate a transmission, the object enable bit (OBJ_EN bit) of the MnCTL register associated with that Transmit Message Object n must be set, except when transmitting an Auto-Acknowledge Frame in CANopen. This will allow this ready-to-transmit message to participate in the pre-arbitration process. In this connection, if more than one message is ready to be transmitted (i.e., if more than one Transmit Message Object is enabled), a Tx Pre-Arbitration process will be performed to determine which enabled Transmit Message Object will be selected for transmission. There are two Tx Pre-Arbitration policies which the user can choose between by setting or clearing the Pre_Arb bit in the GCTL register.

After a Tx Message Complete interrupt is generated in response to a determination being made by the message handler that a completed message has been successfully transmitted, the Tx Pre-Arbitration process is "reset", and begins again. Also, if the "winning" Transmit Message Object subsequently loses arbitration on the CAN bus, the Tx Pre-Arbitration process gets reset and begins again. If there is only one Transmit Message Object whose OBJ_EN bit is set, it will be selected regardless of the Tx Pre-Arbitration policy selected.

Once an enabled Transmit Message Object has been selected for transmission, the DMA engine **38** will begin retrieving the transmit message data from the message buffer associated with that Transmit Message Object, and will begin transferring the retrieved transmit message data to the CCB **42** for transmission. The same DMA engine and address pointer logic is used for message retrieval of transmit messages as is used for message storage of receive messages, as described previously. Further, message buffer location and size information is specified in the same way, as described previously. In short, when a transmit message is retrieved, it will be written by the DMA engine **38** to the CCB **42** sequentially. During this process, the DMA engine **38** will keep requesting the bus; when bus access is granted, the DMA engine **38** will sequentially read the transmit message data from the location in the message buffer currently pointed to by the address pointer logic; and, the DMA engine **38** will sequentially write the retrieved transmit

message data to the CCB **42**. It is noted that when preparing a message for transmission, the user application must not include the CAN ID and Frame Information fields in the transmit message data written into the designated message buffer, since the Transmit (Tx) logic will retrieve this information directly from the appropriate MnMIDH, MnMIDL, and MnMSKH registers.

The XA-C3 microcontroller **20** does not handle the transmission of fragmented messages in hardware. It is the user's responsibility to write each CAN Frame of a fragmented message to the appropriate message buffer, enable the associated Transmit Message Object for transmission, and wait for a completion before writing the next CAN Frame of that fragmented message to the appropriate message buffer. The user application must therefore transmit multiple CAN Frames one at a time until the whole multi-frame, fragmented transmit message is successfully transmitted. However, by using multiple Transmit Message Objects whose object numbers increase sequentially, and whose CAN IDs have been configured identically, several CAN Frames of a fragmented transmit message can be queued up and enabled, and then transmitted in order.

To avoid data corruption when transmitting messages, there are three possible approaches:

1. If the Tx Message Complete interrupt is enabled for the transmit message, the user application would write the next transmit message to the designated transmit message buffer upon receipt of the Tx Message Complete interrupt. Once the interrupt flag is set, it is known for certain that the pending transmit message has already been transmitted.
2. Wait until the OBJ_EN bit of the MnCTL register of the associated Transmit Message Object clears before writing to the associated transmit message buffer. This can be accomplished by polling the OBJ_EN bit of the MnCTL register of the associated Transmit Message Object.
3. Clear the OBJ_EN bit of the MnCTL register of the associated Transmit Message Object while that Transmit Message Object is still in Tx Pre-Arbitration.

In the first two cases above, the pending transmit message will be transmitted completely before the next transmit message gets transmitted. For the third case above, the transmit message will not be transmitted. Instead, a transmit message with new content will enter Tx Pre-Arbitration.

There is an additional mechanism that prevents corruption of a message that is being transmitted. In particular, if a transmission is ongoing for a Transmit Message Object, the user will be prevented from clearing the OBJ_EN bit in the MnCTL register associated with that particular Transmit Message Object.

CAN/CAL RELATED INTERRUPTS

The CAN/CAL module **77** of the XA-C3 microcontroller **20** is presently configured to generate the following five different Event interrupts to the XA CPU Core **22**:

1. Rx Message Complete
2. Tx Message Complete
3. Rx Buffer Full
4. Message Error
5. Frame Error

For single-frame messages, the "Message Complete" condition occurs at the end of the single frame. For multi-frame (fragmented) messages, the "Message Complete" condition occurs after the last frame is received and stored. Since the XA-C3 microcontroller **20** hardware does not recognize or handle fragmentation for transmit messages, the Tx Message Complete condition will always be generated at the end of each successfully transmitted frame.

As previously mentioned, there is a control bit associated with each Message Object indicating whether a Message Complete condition should generate an interrupt, or just set a "Message Complete Status Flag" (for polling) without generating an interrupt. This is the INT_EN bit in the MnCTL register associated with each Message Object n.

There are two 16-bit MMRs **40**, MCPLH and MCPLL, which contain the Message Complete Status Flags for all 32 Message Objects. When a Message Complete (Tx or Rx) condition is detected for a particular Message Object, the corresponding bit in the MCPLH or MCPLL register will be set. This will occur regardless of whether the INT_EN bit is set for that particular Message Object (in its associated MnCTL register), or whether Message Complete Status Flags have already been set for any other Message Objects.

In addition to these 32 Message Complete Status Flags, there is a Tx Message Complete Interrupt Flag and an Rx Message Complete Interrupt Flag, corresponding to bits [1] and [0], respectively, of an MMR **40** designated CANINTFLG, which will generate the actual Event interrupt requests to the XA CPU Core **22**. When an End-of-Message condition occurs, at the same moment that the Message Complete Status Flag is set, the appropriate Tx or Rx Message Complete Interrupt flip-flop will be set provided that INT_EN=1 for the associated Message Object, and provided that the interrupt is not already set and pending.

Further details regarding the generation of interrupts and the associated registers can be found in the XA-C3 Functional Specification and in the XA-C3 CAN Transport Layer Controller User Manual, both of which are part of the parent Provisional Application Serial No. 60/154,022, the disclosure of which has been fully incorporated herein for all purposes.

MESSAGE BUFFERS

As was previously described in detail hereinabove, the XA-C3 microcontroller **20** supports up to 32 separate and independent Message Objects, each of which is set-up or defined by virtue of the user (programmer) configuring (programming) some or all of the eight MMRs **40** dedicated to that Message Object. In the XA-C3 microcontroller **20**, each of the 32 Message Objects is assigned its own block of address space in data memory, which serves as its message buffer for data storage. The size and location of each message buffer is programmable, and thus, reconfigurable "on the fly" by the user/programmer. The message buffers can be positioned in any desired location within the overall data memory space addressable by the XA-C3 microcontroller **20**, which is presently configured to be a 16 Mbyte overall memory space. These message buffers can be located in the XRAM **28** and/or in any off-chip portion of the overall data memory space.

The location of the message buffer associated with each Message Object n is established by programming the MMR **40** designated MnBLR associated with that Message Object, i.e., by programming the Message n Buffer Location Register. The size of the message buffer associated with each Message Object is established by programming the MMR **40** designated MnBSZ associated with that Message Object, i.e., by programming the Message n Buffer SiZe Register. In the XA-C3 microcontroller **20**, allowable buffer sizes are 2, 4, 8, 16, 32, 64, 128, or 256 bytes. Users can select the size

of each message buffer based on the anticipated length of the incoming message, or they can conserve memory by deliberately specifying smaller buffers at the expense of increased processor intervention to handle more frequent buffer-full conditions. In the XA-C3 microcontroller **20**, Direct Memory Access (DMA) (i.e., the DMA engine **38**) is used to enable the XA-C3 CAN/CAL module **77** to directly access the 32 message buffers without interrupting the XA-C3 processor (CPU) core **22**.

The XA-C3 CAN/CAL module **77** uses the values programmed into the buffer size registers MnBSZ to reserve the designated number of bytes of storage for each Message Object n. For Receive Message Objects, this field is also used by logic in the XA-C3 CAN/CAL module **77** to calculate the total number of bytes that have actually been stored in the message buffers, and to identify when a buffer-full condition is reached. Each time a byte of data is stored in a message buffer associated with a Message Object n, the XA-C3 CAN/CAL module **77** concurrently accesses the MnBSZ and MnBLR registers associated with that Message Object. Logic incorporated within the XA-C3 CAN/CAL module **77** decodes the buffer size for that Message Object and compares the decoded buffer size to the address pointer to determine current byte count and available space left in that Message Object's message buffer.

The present implementation of the XA-C3 microcontroller **20** requires that all of the 32 message buffers reside within the same 64 K byte memory segment (or "page"). The user may position the message buffers within any of the 256 pages in the overall XA-C3 data memory space (i.e., 256×64 Kbytes=16 M bytes). Programming the locations of the message buffers is accomplished in two steps.

The first step is to program the page number in which all of the message buffers reside into the MMR **40** designated as the MBXSR register, which is one of the CCB Registers depicted in FIG. **4**. As was previously described, the contents of this register are subsequently used as the eight MSBs of address for all DMA accesses to any of the message buffers. This register also establishes the memory page in which the XRAM **28** resides.

The second step is to program the base address (16 bits) for each individual message buffer into the MnBLR associated with that message buffer. These 16-bit address values initially specified by the user/programmer constitute the base addresses of the 32 respective message buffers within the 64 K byte memory page specified in the MBXSR register for all message buffers. It should be noted that the message buffers can be placed apart from one another, as there is no requirement that the message buffer space be continuous (i.e., that the message buffers reside in physically contiguous locations within the data memory space). Further, it should also be noted that some or all of the message buffers can be placed in off-chip memory, and others in the on-chip XRAM **28**. In the XA-C3 microcontroller **20**, it is required that each message buffer start at a binary boundary for its size (i.e., the 8 LSBs must be zero for a 256-byte message buffer, the 7 LSBs must be zero for a 128-byte message buffer, etc.).

DMA access to each of the message buffers is achieved by using the 8 bits stored in the MBXSR register as the 8 MSBs of the address of that message buffer, and the 16 bits stored in the MnBLR register for that message buffer as the 16 LSBs of the address of that message buffer. The base address initially programmed by the user into the MnBLR register for that message buffer is the address of the first (bottom) location of that message buffer. When the first frame of a new receive message arrives, the CAN/CAL module **77** hardware writes a semaphore code into this bottom location before beginning to store actual data bytes, starting at the next location in that message buffer. At the end of the new receive message (or when a buffer-full condition is detected), the CAN/CAL module **77** hardware computes the total number of bytes actually stored in that message buffer, and writes this value into the bottom location of that message buffer. The processor (i.e., the XA CPU Core **22**) can then read this value and determine precisely how many additional bytes must be read and processed.

Each time a new byte of data must be written to (for receive messages) or retrieve from (for transmit messages) a message buffer, the DMA engine **38** reads the MnBLR register for that message buffer in order to retrieve the current address pointer for the associated Message Object. The DMA engine **38** concatenates the 8 MSBs stored in the global Message Buffer Segment Register (i.e., the MBXSR register) and the 16 LSBs stored in the MnBLR register for that message buffer to form a complete 24-bit message buffer address. The DMA engine **38** then passes this address to the Memory Interface (MIF) unit **30**, along with a flag indicating that the DMA engine **38** requires access to the memory. As soon as the current set of XA-C3 processor memory accesses are completed, the MIF unit **30** will initiate a memory read or write to the address provided by the DMA engine **38**, and then permit the DMA engine **38** to perform the required data transfer to/from the desired message buffer. DMA accesses are typically done two bytes at a time (i.e., as a 16-bit operation). However, 8-bit operations are employed when there is only a single byte to be transferred.

As soon as the requested DMA operation is completed, the DMA engine **38** increments the 16-bit address value stored in the MnBLR register associated with that message buffer (by one or two, depending upon whether a one byte or two byte access was performed), and writes this value back into the MnBLR register for that message buffer. Thus, the MnBLR registers, along with the associated increment logic within the DMA engine **38**, effectively function as a set of 32 binary "counters". Thus, at any given time, each MnBLR register contains the address which will be used for the next data access to the message buffer associated with the Message Object n. In this manner, the MnBLR register for each message buffer serves as an address-pointer. These address-pointer fields are also readable at any time by the processor under software control.

The above-described approach to message storage also provides an extremely quick and efficient means of freeing up a message buffer when a message completes or when a message buffer is full. The software can respond to a message-complete interrupt or a buffer-full interrupt by simply repositioning the message-buffer space for that particular Message Object to somewhere else in the message buffer memory space. This is accomplished by performing a single write operation to modify the buffer base-address specified in the appropriate MnBLR register (i.e., "address-pointer"). This is essentially the extent of a very short interrupt handling routine. These interrupts must be handled quickly because the message buffer must be freed-up for subsequent message reception. Interrupt response is particularly critical if many completed messages are stacked up and need to be dealt with at once. Once this buffer repositioning is accomplished, the hardware is immediately ready to receive a new message over that Message Object "channel" (or, the continuation of the current message, in the case of a buffer-full interrupt). The memory space that was previously designated as the message buffer for that Message Object n

still contains the previously-received message data, but this space now becomes just part of the long-term data memory space. The message information stored in this long-term data memory space can then be processed by the software at its leisure.

This same buffer repositioning technique can be employed for Transmit Messages to facilitate fragmentation. Unlike the receive case, the XA-C3 CAN/CAL Module **77** does not automatically assemble fragmented outgoing messages. It is incumbent upon the software to "load" a new message frame each time the previous frame is transmitted. Using the XA-C3 microcontroller 20 message storage scheme, however, the software can construct an entire fragmented message prior to enabling transmission. As each frame is transmitted, the processor (XA CPU Core **22**) only needs to reposition the buffer (again, using a single write operation) to point to the location of the next frame. This is much faster than competing devices, which require the processor to move up to 13 bytes of data from memory to a dedicated transmit buffer.

It will be appreciated that with the above-described message buffer scheme of the present invention, each message buffer can be regarded as a separate FIFO having an independently programmable buffer length, which provides a revolutionary approach to storing sequential messages of varying lengths without any CPU intervention.

THE PRESENT INVENTION

As described hereinabove, each incoming (received) CAN Frame that passes Acceptance Filtering will be automatically stored, via the DMA engine **38**, into the message buffer for the Receive Message Object that particular CAN Frame was found to have matched, without interrupting the XA CPU Core **22**. Under the CAN protocol, if a Message Object is an enabled Receive Message Object, and its associated MnCTL register has its FRAG bit set to '1' (i.e., automatic fragmented message assembly is enabled for that particular Receive Message Object), then the CAN Frames that match that particular Receive Message Object will be stored sequentially in the message buffer for that particular Receive Message Object using the format shown in FIG. **12**.

When writing message data into a message buffer associated with a Message Object n, the DMA engine **38** will generate addresses automatically starting from the base address of that message buffer (as specified in the MnBLR register associated with that Message Object n). Since the size of that message buffer is specified in the MnBSZ register associated with that Message Object n, the DMA engine **38** can determine when it has reached the top location of that message buffer. If the DMA engine **38** determines that it has reached the top location of that message buffer, and that the message being written into that message buffer has not been completely transferred yet, the DMA engine **38** will wrap around by generating addresses starting from the base address of that message buffer again. Some time before this happens, a warning interrupt will be generated so that the user application can take the necessary action to prevent data loss.

For single-frame messages, the "Message Complete" condition occurs at the end of the single frame. For multi-frame (fragmented) messages, the "Message Complete" condition occurs after the last frame is received and stored. Since the XA-C3 microcontroller **20** hardware does not recognize or handle fragmentation for transmit messages, the Tx Message Complete condition will always be generated at the end of each successfully transmitted frame.

As previously mentioned, there is a control bit associated with each Message Object indicating whether a Message Complete condition should generate an interrupt, or just set a "Message Complete Status Flag" (for polling) without generating an interrupt. This is the INT_EN bit in the MnCTL register associated with each Message Object n.

There are two 16-bit MMRs **40**, MCPLH and MCPLL, which contain the Message Complete Status Flags for all 32 Message Objects. When a Message Complete (Tx or Rx) condition is detected for a particular Message Object, the corresponding bit in the MCPLH or MCPLL register will be set. This will occur regardless of whether the INT_EN bit is set for that particular Message Object (in its associated MnCTL register), or whether Message Complete Status Flags have already been set for any other Message Objects. These two status registers (MCPLH and MCPLL) are readable at any time by the XA CPU Core **22**, thus providing a "polling" capability. Each bit in these registers can be cleared by the software once the corresponding message has been processed.

In addition to these 32 Message Complete Status Flags, there is a Tx Message Complete Interrupt Flag and an Rx Message Complete Interrupt Flag, corresponding to bits [1] and [0], respectively, of an MMR **40** designated CANINTFLG, which will generate the actual Event interrupt requests to the XA CPU Core **22**. When an End-of-Message condition occurs, at the same moment that the Message Complete Status Flag is set, the appropriate Tx or Rx Message Complete Interrupt flip-flop will be set provided that INT_EN=1 for the associated Message Object, and provided that the interrupt is not already set and pending.

Further, the MMR **40** designated MCIR (Message Complete Info Register) in FIG. **4**, contains six bits, including five bits that identify the lowest-numbered interrupt-enabled Message Object for which an End-of-Message condition exists, and one bit that indicates whether an End-of-Message condition exists for any interrupt-enabled Message Object. The Message Complete Info Register allows the XA CPU Core **22** to directly read which object(s) currently have a message-complete interrupt pending. This register (MCIR) is updated on every clock edge, so that it continuously identifies the number of the lowest-numbered, interrupt-enabled Message Object for which an End-of-Message condition exists.

In further accordance with the present invention, the message handler logic within the CAN/CAL module **77** constantly (every clock cycle) monitors the output of the Message Complete Status Flag Registers (MCPLH and MCPLL), along with the 32 interrupt-enable bits from the 32 Message Object Control Registers (MnCTL), in order to identify the lowest-numbered, interrupt-enabled Message Object for which an End-of-Message condition exists. Every clock cycle, this object number is loaded into the Message Complete Info Register (MCIR). Thus, any changes in the Status Flags, e.g., due to a new message completing, or due to a bit being cleared by software, is immediately reflected in the Message Complete Info Register.

If the XA CPU Core **22** is able to respond to a Message-Complete Interrupt right away, there should be only one completed message pending (excluding any messages for which interrupt generation has been disabled). The software can read the Message Complete Info Register to determine which Message Object has a completed message, process that message, clear the appropriate Status Flag within the Message Complete Status Flag Registers (MCPLH and

MCPLL), clear the Rx Message Complete Interrupt Flag corresponding to bit [0] of the MMR **40** designated CANINTFLG, and then return to its previous state.

In further accordance with the present invention, when the XA CPU Core **22** is interrupted by a Tx or Rx message-complete event, it will first read the Message Complete Info Register to determine which Message Object has a completed message. It will then process that message and then clear the Message-Complete Status Flag corresponding to that Message Object. At this point, the XA-C3 CPU Core **22** will again read the Message Complete Info Register to determine whether there are any additional, interrupt-enabled Message Object for which an End-of-Message condition exists.

If the XA CPU Core **22** was able to respond to the Message-Complete Interrupt right away and process it quickly, there are unlikely to be any additional Message-Complete Interrupts pending. In this case, the XA CPU Core **22** can clear the Rx Message Complete Interrupt Flag, and return to its previous state.

If, on the other hand, the XA CPU Core **22** took a while to handle the first Message-Complete Interrupt, additional messages may very well have completed in the meantime. In this case, a new message-completed Message Object will have "popped" to the top as soon as the software cleared the Message-Complete Status Flag corresponding to the first message-completed Message Object. The number of this Message Object will now appear in the Message Complete Info Register. The software will then process this newly-completed message, clear the appropriate Message-Complete Status Flag, and once again read the Message Complete Info Register to determine whether there are any additional completed messages which require processing. Of course, this process will be repeated until there are no more completed messages remaining to be processed. At this point, the software can clear the Rx Message-Complete Interrupt Flag and return to its previous state.

It will be appreciated by those skilled in the pertinent art that the reason the Message Complete Info Register is so important is that it would be an extremely cumbersome and time-consuming task for the software to isolate the first bit set in the 32-bit status field and translate this into a Message Object number for processing. The Message Complete Info Register allows the software to read this value directly, and process each completed message sequentially. The 32 bits constituting the Message-Complete Status Flags are still readable by the XA CPU Core **22** for polling purposes. Thus, if the software simply wants to determine whether a particular Message Object has a completed message, it is a very simple task to just test the appropriate one of the Message-Complete Status Flags.

The specific implementation of the controlling logic within the XA-C3 microcontroller **20** is as follows. The Message Management engine within the CAN/CAL module **77** provides the following signals to a CAN Interrupt logic module within the Message Handling engine of the CAN/CAL module:

1. 5-bit Rx_Object_Number: Indicates the number of the current Receive Message Object.
2. 5-bit Tx_Object_Number: Indicates the number of the current Transmit Message Object.
3. Rx_Complete Flag: Indicates a message-complete (End-of-Message) condition for the current Receive Message Object.
4. Tx_Complete Flag: Indicates a message-complete (End-of-Message) condition for the current Transmit Message Object.
5. Rx_Object_Ena: Single-bit flag indicating whether the current Receive Message Object is enabled to generated End-of-Message Interrupts.
6. Tx_Object_Ena: Single-bit flag indicating whether the current Transmit Message Object is enabled to generated End-of-Message Interrupts.
7. 32 Individual Object Interpt-Ena bits directly from the respective individual Message Object Control Registers (MnCTLs).

The CAN Interrupt module processes these signals as follows:

The Rx_Complete Flag and the Tx_Complete Flag are used to control a 5-input, 2:1 multiplexer which selects between either the Rx_Object_Number of the Tx_Object_Number, depending upon which type of message has completed. This value goes through a 5:32 decoder, the outputs of which are routed to the 'D' inputs of the two 16-bit Message Complete Status Flag Registers (MCPLH and MCPLL). These registers are enabled to be updated whenever either the Rx_Complete Flag or the Tx_Complete Flag is active. Again, these flags indicate whether any message has been completed, regardless of whether the corresponding Message Object has been interrupt-enabled.

The Rx_Complete and Tx_Complete Interrupt Flags are generated whenever the corresponding Rx_Complete Flag or Tx_Complete Flag is active and the appropriate Object_Ena bit is high. Since Rx_Object_Ena and Tx_Object_Ena are single bits reflecting the interrupt-enabled/disabled status for the particular Message Object being processed at the time the Interrupt Flag gets set, the only logic required for generating the Rx_Complete and Tx_Complete Interrupt Flags is a simple AND gate.

Each of the 32 output bits of the 32 Message Complete Status Flag Registers (MCPLH and MCPLL) is logically AND'ed with the Intrpt-Ena bit for the corresponding Message Object. The outputs of these 32 AND gates (logic AND operations) are routed to a priority encoder that determines the first one in the sequence to exhibit a logic "1" state, and then encodes the Object Number corresponding to that bit. This Object Number is loaded into the Message Complete Info Register on the next clock edge. The logical OR of these 32 AND gates is loaded into the 6^{th} bit position of the Message Complete Info Register to indicated whether there are any interrupt-enabled Message Objects for which an End-of-Message condition exists that have not yet been processed.

Although the present invention has been described in detail hereinabove in the context of a specific preferred embodiment/implementation, it should be clearly understood that many variations, modifications, and/or alternative embodiments/implementations of the basic inventive concepts taught herein which may appear to those skilled in the pertinent art will still fall within the spirit and scope of the present invention, as defined in the appended claims.

What is claimed is:

1. A CAN microcontroller that supports a plurality of uniquely-numbered message objects, comprising:

a processor core that runs CAN applications;

a plurality of message buffers associated with respective ones of the message objects;

a plurality of individual message object registers associated with each message object, including at least one control register that contains an interrupt-enable control bit, a receive enable bit, and a transmit enable bit;

a plurality of global message object control registers, including:

at least one message complete status register that contains a plurality of status flag bits for respective ones of the message objects;

at least one interrupt flag register that contains a receive complete interrupt flag bit and a transmit complete interrupt flag bit; and,

a message complete info register that contains a plurality of message object identification bits and a status bit;

a CAN/CAL module that automatically assembles fragmented, multi-frame messages, wherein the CAN/CAL module includes:

an acceptance filtering function that performs acceptance filtering on each incoming, multi-frame message by comparing a screener field of the incoming, multi-frame message with an acceptance filter field associated with each message object which has its associated receive enable bit set, wherein the incoming, multi-frame message is accepted if its screener field matches the acceptance filter field of a receive-enabled message object;

a message handling function that automatically transfers successive frames of an accepted incoming multi-frame message to the message buffer associated with the matching receive-enabled message object;

an end-of-message detection function that detects an end-of-message condition which occurs when the last frame of the accepted incoming multi-frame message has been stored in the message buffer associated with the matching receive-enabled message object; and,

an end-of-message detection handling and interrupt generation function that, in response to the detection of the end-of-message condition:

sets the status flag bit contained in the at least one message complete status register corresponding to the matching receive-enabled message object;

sets the receive complete interrupt flag bit contained in the at least one interrupt flag register, if the interrupt-enable control bit contained in the at least one control register associated with the matching receive-enabled message object is set; and,

sets the status bit contained in the message complete info register, if the interrupt-enable control bit contained in the at least one control register associated with the matching receive-enabled message object is set.

2. The CAN microcontroller as set forth in claim **1**, further comprising a data memory space, the plurality of message buffers being located in the data memory space.

3. The CAN microcontroller as set forth in claim **1**, wherein the CAN/CAL module includes a DMA engine that facilitates direct transfers of message data to the message buffers without interrupting the processor core.

4. The CAN microcontroller as set forth in claim **1**, wherein the individual message object registers and the global message object control registers comprise memory-mapped registers.

5. The CAN microcontroller as set forth in claim **1**, further comprising:

a data memory space; and,

wherein the individual message object registers and the global message object control registers comprise memory-mapped registers that are mapped to a respective portion of the data memory space.

6. The CAN microcontroller as set forth in claim **1**, wherein the plurality of message buffers are located in the data memory space.

7. The CAN microcontroller as set forth in claim **1**, wherein the at least one control register associated with each message buffer is programmable for the purpose of enabling or disabling the interrupt-enable control bit, the receive enable bit, and the transmit enable bit.

8. The CAN microcontroller as set forth in claim **1**, wherein the end-of-message detection handling and interrupt generation function:

monitors the status of the status flag bits contained in the at least one global message object control register, and the status of the interrupt-enable control bits contained in the individual message object registers, in order to identify the lowest-numbered interrupt-enabled message object whose associated status flag bit is set; and,

sets the plurality of message object identification bits contained in the message complete info register to reflect the object number of the lowest-numbered interrupt-enabled message object whose associated status flag bit is set.

9. The CAN microcontroller as set forth in claim **1**, wherein the end-of-message detection handling and interrupt generation function monitors the status of the status flag bits contained in the at least one global message object control register, and the status of the interrupt-enable control bits contained in the individual message object registers, every clock cycle of a system clock.

10. The CAN microcontroller as set forth in claim **1**, wherein a current application running on the processor core checks the status of the status flag bits contained in the at least one message complete status register, at selected times.

11. The CAN microcontroller as set forth in claim **1**, wherein a current application running on the processor core checks the status of the status bit contained in the message complete info register to determine whether or not there are any pending completed messages associated with a respective interrupt-enabled message object.

12. The CAN microcontroller as set forth in claim **11**, wherein, in response to a determination that there is a pending completed message based on the status of the status bit contained in the message complete info register, the current application running on the processor core:

processes the completed message corresponding to the lowest-numbered receive-enabled message object identified by the message object identification bits contained in the message complete info register;

clears the status flag bit contained in the at least one control register associated with the lowest-numbered receive-enabled message object;

checks the status of the status bit contained in the message complete info register; and,

repeats each of the above-recited operations if the status bit contained in the message complete info register is enabled, until the status flag bit is no longer enabled.

13. The CAN microcontroller as set forth in claim **1**, wherein the CAN/CAL module generates a message-complete interrupt in response to detection of an end-of-message condition if the interrupt-enable control bit contained in the at least one control register associated with the matching receive-enabled message object is enabled.

14. The CAN microcontroller as set forth in claim **10**, wherein the current application running on the processor core processes the completed message corresponding to the message object associated with an enabled status flag bit that is contained in the at least one message complete status register.

15. The CAN microcontroller as set forth in claim 13, wherein the current application running on the processor core processes the completed message corresponding to the matching, interrupt-enabled, receive-enabled message object, in response to the message-complete interrupt.

16. A CAN microcontroller that supports a plurality of uniquely-numbered message objects, comprising:

a processor core that runs CAN applications;

a plurality of message buffers associated with respective ones of the message objects;

a plurality of individual message object registers associated with each message object, including at least one control register that contains an interrupt-enable control bit, a transmit enable bit, and a receive enable bit;

a plurality of global message object control registers, including:

at least one message complete status register that contains a plurality of status flag bits for respective ones of the message objects;

at least one interrupt flag register that contains a receive complete interrupt flag bit and a transmit complete interrupt flag bit; and,

a message complete info register that contains a plurality of message object identification bits and a status bit;

a CAN/CAL module that automatically assembles fragmented, multi-frame messages, wherein the CAN/CAL module includes:

a message handling function that automatically transfers successive frames of an incoming multi-frame message to the message buffer associated with a corresponding message object;

an end-of-message detection function that detects an end-of-message condition which occurs when the last frame of the accepted incoming multi-frame message has been stored in the message buffer associated with the corresponding message object; and,

an end-of-message detection handling and interrupt generation function that, in response to the detection of the end-of-message condition:

sets the status flag bit contained in the at least one message complete status register associated with the corresponding message object;

sets the receive complete interrupt flag bit contained in the at least one interrupt flag register, if the interrupt-enable control bit contained in the at least one control register associated with the corresponding message object is set; and,

sets the status bit contained in the message complete info register, if the interrupt-enable control bit contained in the at least one control register associated with the corresponding message object is set.

17. The CAN microcontroller as set forth in claim 16, wherein the at least one control register associated with each message buffer is programmable for the purpose of enabling or disabling the interrupt-enable control bit, the receive enable bit, and the transmit enable bit.

18. The CAN microcontroller as set forth in claim 16, wherein the end-of-message detection handling and interrupt generation function:

monitors the status of the status flag bits contained in the at least one global message object control register, and the status of the interrupt-enable control bits contained in the individual message object registers, in order to identify the lowest-numbered interrupt-enabled message object whose associated status flag bit is set; and,

sets the plurality of message object identification bits contained in the message complete info register to reflect the object number of the lowest-numbered interrupt-enabled message object whose associated status flag bit is set.

19. The CAN microcontroller as set forth in claim 16, wherein the end-of-message detection handling and interrupt generation function monitors the status of the status flag bits contained in the at least one global message object control register, and the status of the interrupt-enable control bits contained in the individual message object registers, every clock cycle of a system clock.

20. The CAN microcontroller as set forth in claim 16, wherein a current application running on the processor core checks the status of the status flag bits contained in the at least one message complete status register, at selected times.

21. The CAN microcontroller as set forth in claim 16, wherein a current application running on the processor core checks the status of the status bit contained in the message complete info register to determine whether or not there are any pending completed messages associated with a respective interrupt-enabled message object.

22. The CAN microcontroller as set forth in claim 21, wherein, in response to a determination that there is a pending completed message based on the status of the status bit contained in the message complete info register, the current application running on the processor core:

processes the completed message corresponding to the lowest-numbered receive-enabled message object identified by the message object identification bits contained in the message complete info register;

clears the status flag bit contained in the at least one control register associated with the lowest-numbered receive-enabled message object;

checks the status of the status bit contained in the message complete info register; and,

repeats each of the above-recited operations if the status bit contained in the message complete info register is enabled, until the status flag bit is no longer enabled.

23. The CAN microcontroller as set forth in claim 16, wherein the CAN/CAL module generates a message-complete interrupt in response to detection of an end-of-message condition if the interrupt-enable control bit contained in the at least one control register associated with the corresponding receive-enabled message object is enabled.

24. The CAN microcontroller as set forth in claim 20, wherein the current application running on the processor core processes the completed message corresponding to the message object associated with an enabled status flag bit that is contained in the at least one message complete status register.

25. The CAN microcontroller as set forth in claim 23, wherein the current application running on the processor core processes the completed message, in response to the message-complete interrupt.

26. A CAN microcontroller that supports a plurality of uniquely-numbered message objects, comprising:

a processor that runs CAN applications;

means for selectively enabling any one or more of the message objects for interrupt generation;

means for monitoring incoming messages to determine when an end-of-message condition occurs;

means for generating a completed message status flag for a particular message object in response to a determi-

nation that an end-of-message condition has occurred with respect to that particular message object;

means for generating a message-complete interrupt to the processor in response to a determination that an end-of-message condition has occurred with respect to any message object that has been interrupt-enabled; and

means for generating an indication of the lowest-numbered message object for which it has been determined an end-of-message condition has occurred.

27. The CAN microcontroller as set forth in claim **26**, further comprising means for generating an indication of the lowest-numbered, interrupt-enabled message object for which it has been determined an end-of-message condition has occurred.

28. The CAN microcontroller as set forth in claim **26**, further comprising means for generating pending message-complete status flag whenever there is at least one message object that has a completed message status flag associated therewith.

29. The CAN microcontroller as set forth in claim **26**, further comprising means for generating pending message-complete status flag whenever there is at least one interrupt-enabled message object that has a completed message status flag associated therewith.

30. The CAN microcontroller as set forth in claim **27**, wherein a current CAN application running on the processor, in response to the message-complete interrupt, processes the completed message corresponding to the lowest-numbered message object for which it has been determined an end-of-message condition has occurred.

31. The CAN microcontroller as set forth in claim **26**, wherein a current CAN application running on the processor, in response to the message-complete interrupt, processes the completed message corresponding to the lowest-numbered message object for which it has been determined an end-of-message condition has occurred.

* * * * *