Application Note

AN1828/D
Rev. 1, 2/2002

Flash Programming via CAN

MOTOROLA
intelligence everywhere™

digital dna™

by Ross McLuckie
East Kilbride, Scotland.

1 Introduction

With the introduction, and growing use of Flash based microcontrollers (MCU), new opportunities exist to extend the capabilities of the Controller Area Network (CAN). One such opportunity would be to use the CAN to examine, modify or reprogram the memory contents of any MCU connected to the network from a single, easily accessible point within the system.

The more traditional methods of providing in-circuit programming of an Electronic Control Module (ECM) are based upon either the Universal Asynchronous Receiver / Transmitter (UART) or an MCU specific interface, such as the single wire interfaces found on Motorola's HC08 (Monitor mode) and HC12 (BDM) products. Using this approach requires dedicated hardware on each ECM and assumes that accessibility to each module is readily available.

From a design point of view, the added cost of dedicated hardware for a diagnostic / development feature and the restrictions placed upon the ECM to meet the accessibility requirements are undesirable to say the least. At this point it easy to understand the benefits of utilizing CAN to provide the desired functionality, each ECM has a CAN connection as part of the standard system, therefore no additional hardware is required, and connection to any node allows communication to all other nodes via CAN.

This concept offers benefits throughout the products' life span, from the development phase through to in-field upgrades, servicing and diagnostic capabilities. During development and testing any module connected to the network could be reprogrammed in-circuit, saving time and effort as well as minimizing the dependencies between product assembly and software development. In-field system upgrades, servicing and diagnostic reports could all be easily achieved, and potential high cost product recalls could be handled much quicker and cheaper with field maintenance.

A considerable amount of additional functionality can be added by implementing some or all of these features, whilst requiring limited effort during the software development cycle.

2 Scope

The purpose of this paper is to focus on the specific features necessary to enable the reader to include the desired functionality into their system. It is assumed the reader is familiar with the use of CAN and Flash memory technology, therefore the discussion will not enter into any great detail on either the CAN specification or device specific Flash programming algorithms. There are numerous other publications which describe in detail the CAN specification, whilst Flash programming algorithms are technology / device specific and although a working example is shown for Motorola's HC12 Flash memory, the principles discussed could be easily extended to any other Flash technology.

3 Objective

It is the intention of this application note to identify and illustrate the key features required, allowing the reader to incorporate the additional functionality, discussed in the introductory section, into their system. In addition to outlining the requirements of the basic 'skeleton' system, some topics will also discuss potential extensions and enhancements that the reader may wish to consider when customizing and tailoring the system to their individual needs.

Although the principles discussed could be applied to any CAN based system incorporating MCUs with either embedded or external Flash memory, for the purpose of illustration, the remainder of this document will describe how to build a demonstration system based on Motorola's M68EVB912BC32.

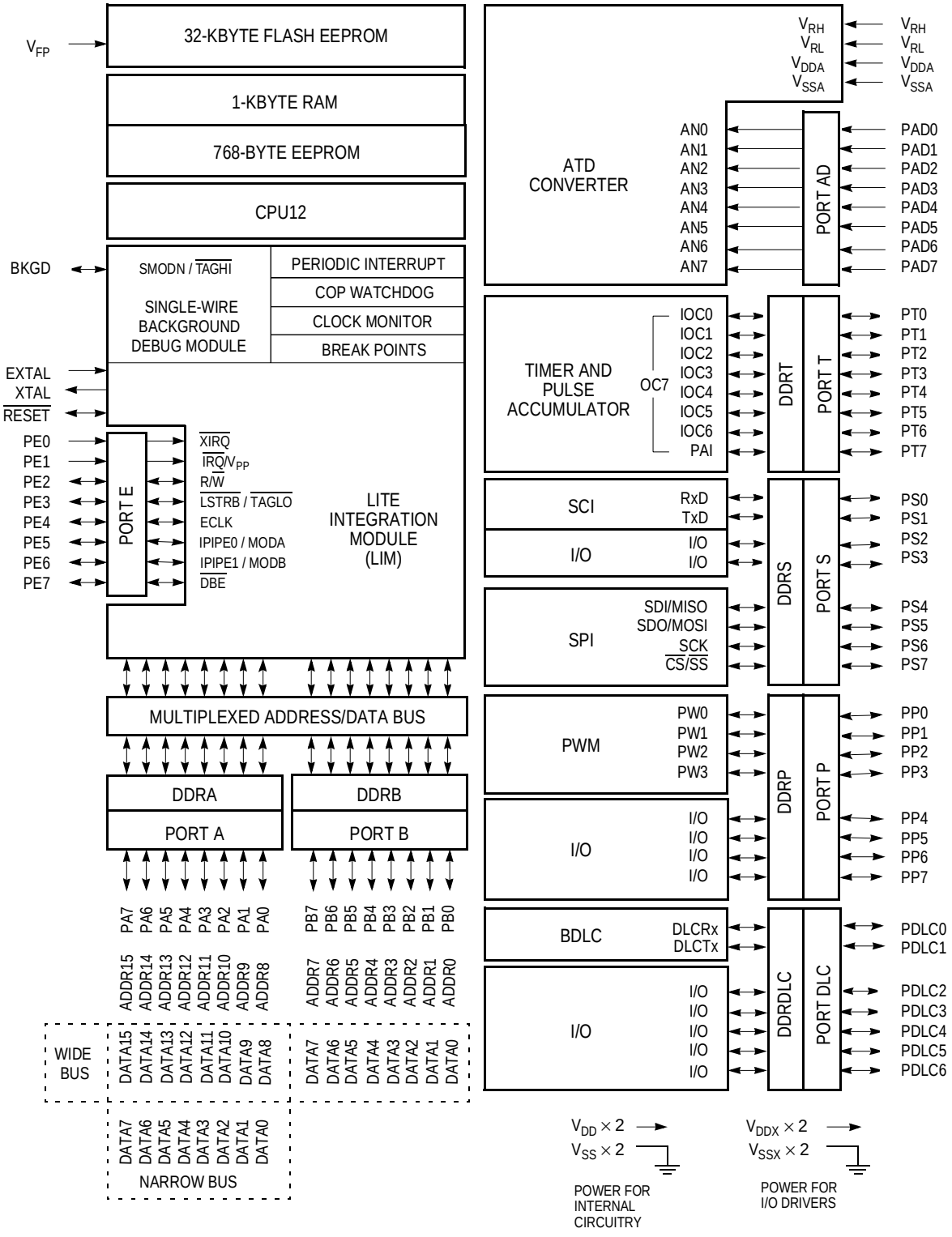

Figure 1. MC68HC912BC32 block diagram

In order to provide this additional functionality, the following key features must be taken into consideration when planning and designing the overall system.

- Provide ‘maintenance’ access to the MCU via the CAN interface.
- Device specific Flash modifying routines.
- A ‘smart cable’ to interface between a PC and the target ECM.
- An API capable of transferring data to the ‘smart cable’.

The following sections will take a closer look at each of these topics and illustrate, through example, the minimum requirements needed to accomplish each task. In addition, each section will discuss ways to enhance and extend the overall performance of the system, allowing the designer to meet their system’s unique requirements.

4 MCU maintenance access via CAN

There is very seldom a single solution to any given design requirement and it is important to determine an appropriate strategy from the offset. For this particular application it would be just as easy to embed the Flash modifying algorithms into the user software and activate them via a CAN message, but this approach comes with many limiting factors. Having Flash algorithms in MCU memory at all times could result in permanent damage if at any time code runaway occurred, less memory would be available for application code and additional functionality would be limited to what was coded in the original application.

A more flexible approach would be to utilize a CAN Load Ram And Execute (LRAE) routine. Flash algorithms would only be loaded into the MCU at the appropriate time, it would be possible to write a very small routine (under 100 bytes) to accomplish the task and only MCU ram size restricts additional functionality.

The basic requirement for the LRAE routine is to implement a CAN protocol which allows data transfer into ram and program execution from ram. Although several CAN protocol definitions already exist, such as CCP (CAN Calibration Protocol), CANopen and SDS™ (Smart Distributed System)[1], for the purpose of demonstration, a simplified custom protocol was adopted.

The flowchart in Figure 2 explains the operation of the LRAE routine, while Table 1 explains how the CAN protocol functions.

1. SDS™ is a trademark of Honeywell Inc.

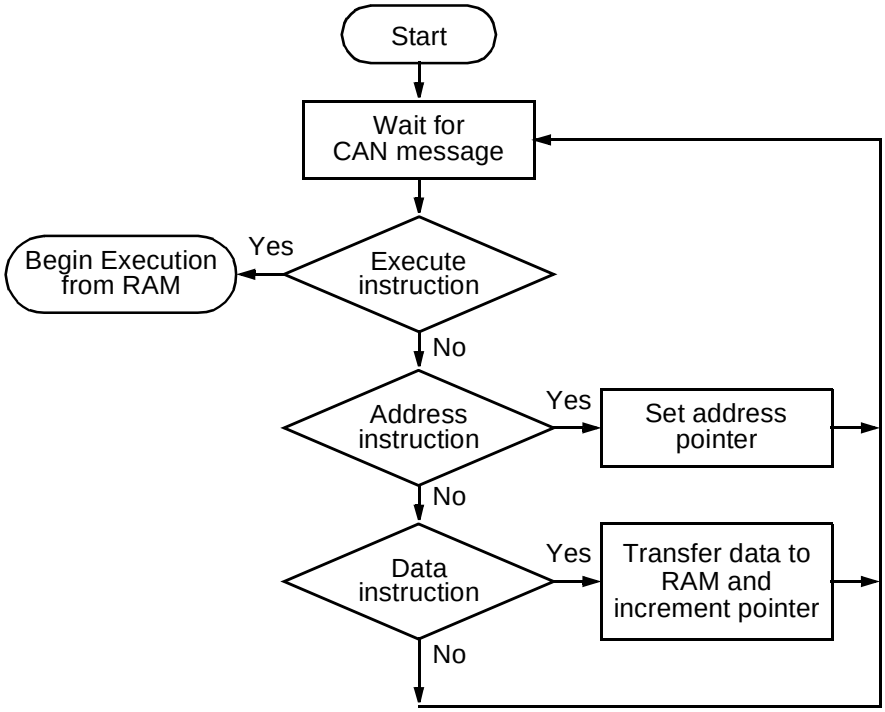

Figure 2. CAN load ram and execute process flow

Table 1. CAN message Rx buffer contents

CAN Rx Buffer	Address Instruction	Data Instruction	Execute Instruction
DSR0	0	2	4
DSR1	Address MSB	Data Byte 1	Address MSB
DSR2	Address LSB	(Data Byte 2)	Address LSB
DSR3	—	(Data Byte 3)	—
DSR4	—	(Data Byte 4)	—
DSR5	—	(Data Byte 5)	—
DSR6	—	(Data Byte 6)	—
DSR7	—	(Data Byte 7)	—

The address instruction initializes a pointer to RAM, where subsequent data bytes are incrementally stored until a new address or execute instruction is received. The data instruction may contain up to seven bytes of data. On receiving the execute instruction, program execution jumps to the address location contained in the execute command.

Listing 1 provides a coded example of how to implement the CAN LRAE protocol shown in Figure 2 and Table 1.

Listing 1 CAN LRAE routine

```
;*******************************************************************************
; A load RAM and execute routine via the CAN network
; Written to run on an MC68HC912BC32
;*******************************************************************************

;*******************************************************************************
; Register definitions
;*******************************************************************************

COPCTL:         EQU    $16
CMCR0:          EQU    $0100

;*******************************************************************************
; Bit definitions for the CMCR0 register
;*******************************************************************************
CSWAI:          EQU    $20
SYNCH:          EQU    $10
TLNKEN:         EQU    $08
SLPAK:          EQU    $04
SLPRQ:          EQU    $02
SFTRES:         EQU    $01
;*******************************************************************************

CBTR0:          EQU    $0102
CRFLG:          EQU    $0104

;*******************************************************************************
; Bit definitions for the CRFLG register
;*******************************************************************************
WUPIF:          EQU    $80
RWRNIF:         EQU    $40
TWRNIF:         EQU    $20
RERRIF:         EQU    $10
TERRIF:         EQU    $08
BOFFIF:         EQU    $04
OVRIF:          EQU    $02
RXF:            EQU    $01
;*******************************************************************************

CTCR:           EQU    $0107
CIDMR0:         EQU    $0114
CIDMR2:         EQU    $0116
RXDSR0:         EQU    $0144
RXDSR1:         EQU    $0145
RXDLR:          EQU    $014C
```

```
;*******************************************************************************
; Standard equates
;*******************************************************************************

StackTop:       EQU     $0BFF
ProtectedBlock: EQU     $FC00
ResetVector:    EQU     $FFFE
CBT:            EQU     $C749
LastInstr:      EQU     $04

                ORG     ProtectedBlock

;*******************************************************************************
; Initialization routine
;*******************************************************************************

lrae:
                LDS     #StackTop           ;initialize stack pointer
                CLR     COPCTL              ;switch off COP watchdog

;*******************************************************************************
; Setup the CAN module
;*******************************************************************************

                BSET    CMCR0,#SFTRES       ;place CAN module in reset
                MOVW    #CBT,CBTR0          ;set up CAN bit timing

                CLR     CTCR
                MOVW    #$FFFF,CIDMR0
                MOVW    #$FFFF,CIDMR2       ;set up module to receive all messages

                BCLR    CMCR0,#SFTRES       ;take CAN module out of reset
canSynch:
                BRCLR   CMCR0,#SYNCH,*      ;synchronize module with CAN bus

;*******************************************************************************
; Wait for CAN message
;*******************************************************************************

waitForMsg:
                BRCLR   CRFLG,#RXF,*        ;wait for CAN message

                BRSET   RXDSR0,#$01         ,waitForNextMsg
                LDAB    RXDSR0
                CMPB    #LastInstr
                BHI     waitForNextMsg      ;ignore invalid instructions

                CLRA                        ;jump to appropriate routine, depending
                JMP     [D,PC]              ;on instruction value
                DC.W    addressInstr        ;(0) initialize RAM pointer
                DC.W    dataInstr           ;(2) load data into RAM
```

```
                DC.W    executeInstr      ;(4) begin execution at given address

;*******************************************************************************
; Setup RAM pointer
;*******************************************************************************

addressInstr:
                LDX     RXDSR1            ;point to RAM address in RXDSR1:2
                BRA     waitForNextMsg

;*******************************************************************************
; Transfer data into RAM
;*******************************************************************************

dataInstr:
                LDAB    RXDLR             ;number of bytes transmitted
                LDY     #RXDSR1           ;start of transmitted data
nextDataByte:
                DECB                      ;ignore command byte
                BEQ     endOfData         ;stop at end of data
                MOVB    1,Y+,1,X+         ;load data into RAM
                BRA     nextDataByte
endOfData:
                BRA     waitForNextMsg

;*******************************************************************************
; Clear CAN Rx flag and begin program execution from new location
;*******************************************************************************

executeInstr:
                MOVB    #RXF,CRFLG        ;clear Rx flag
                LDX     RXDSR1
                JMP X                     ;begin program execution from RXDSR1:2

;*******************************************************************************
; Clear CAN Rx flag and wait for next message
;*******************************************************************************

waitForNextMsg:
                MOVB    #RXF,CRFLG        ;clear Rx flag
                BRA     waitForMsg

;*******************************************************************************
; Define reset vector
;*******************************************************************************

                ORG     ResetVector
                DCW     lrae

;*******************************************************************************
```

Since the LRAE function provides the platform from which the complete system is built upon, then it is also the area where most of the system enhancements and extensions should be added.

Implementing a specific CAN protocol could provide additional functionality, whilst adding a level of security, through message handshaking when establishing a connection to the target MCU. If required, a more complex, custom handshaking protocol could be added, in an attempt to prevent any unauthorized access to the MCU.

The coded example, shown in Listing 1, accepts all CAN messages and is intended for use in a point to point (2-node network) application only. By either adopting a more complex protocol, or utilizing a dedicated CAN filter / identifier for each node on the network, a multiple node network could be easily supported.

From Table 1, it can be seen that the LRAE example sends both the instruction ID and its associated data in the CAN data segment registers (DSR0-7). If it was required to optimize the bandwidth of the CAN bus, the instruction ID could be embedded into the CAN identifier, allowing transmission of up to eight data bytes at a time. The potential improvement on system performance depends upon the number of possible instruction ID’s and the overall size of the data transfer.

It is important to consider the effect of a system failure whilst attempting to modify Flash memory, if for example, a power failure occurred after erasing, but before programming was complete, then the end result would be an erased or partially programmed target MCU. The resultant ‘dead’ node would most likely have to be replaced, which may require significant cost and effort. There are several approaches that could be undertaken to prevent, or at least minimize the risk from this kind of failure. An auxiliary power supply could be incorporated into the maintenance equipment, utilizing the protected memory area of the MCU in order to guarantee a minimal functionality, such as the CAN LRAE routines or in the case of the HC12, provide an appropriate BDM interface. Although it is not possible to eliminate the risk of this type failure entirely, the amount of preventative action taken should depend upon the potential consequences arising from a failure of this nature.

There are many more topics that could be discussed here, and the required functionality will vary from system to system, but it is important to realize when defining the specification for the LRAE routine, the part it plays in limiting the overall system.

5 Device specific Flash modifying, via CAN, routines

With the exception of a few minor differences, the basic requirement for the Flash modifying routine is the same as the LRAE routine. A CAN protocol capable of transferring data into a ram buffer and the ability to both erase and program Flash. Despite the fact that there are numerous Flash technologies, even Motorola's HC08 and HC12 products have different Flash modules, the same basic principles are applicable to them all. The coded example that follows was written for use with the MC68HC912BC32 Flash module.

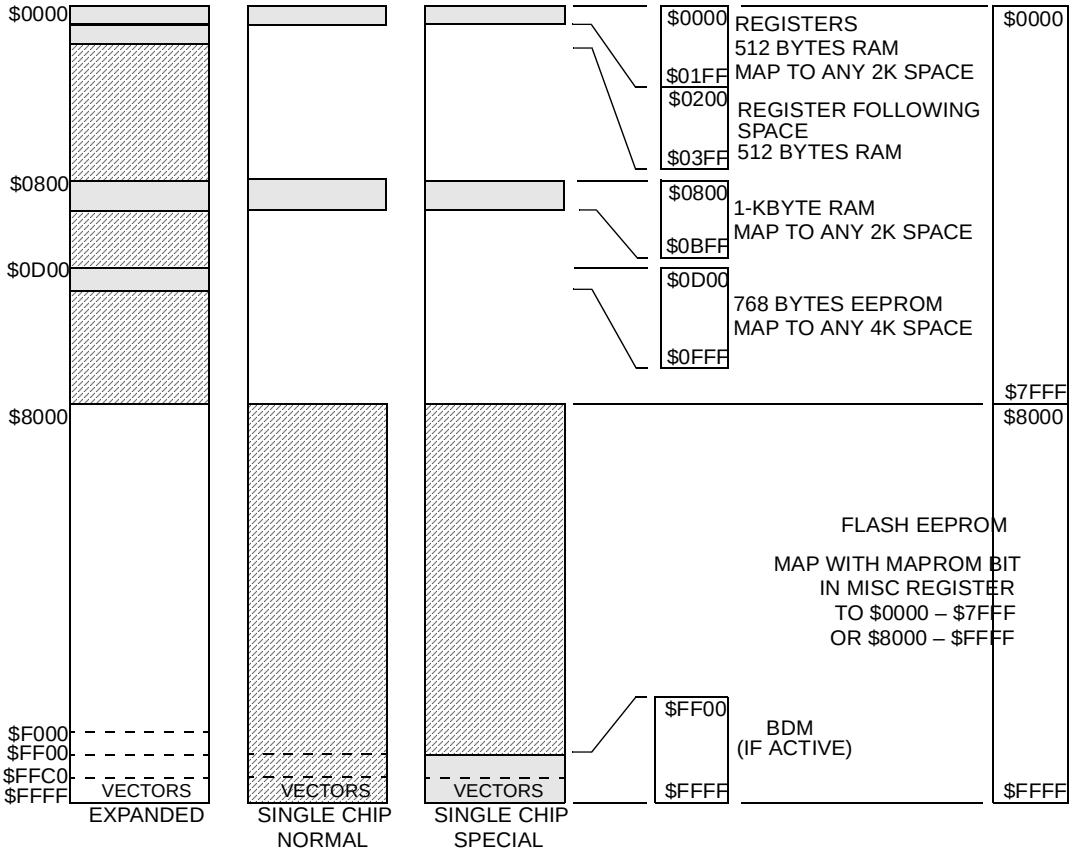

Figure 3. Motorola's MC68HC912BC32 memory map

The biggest difference in functionality from the LRAE routine is the introduction of CAN message handshaking, which gives the target MCU the ability to return status messages after each command request. Although not an essential requirement, the ability to return status information greatly increases the capabilities of the overall system.

An initial status message is sent to indicate that the Flash modifying routines are now running and have control of the target MCU, subsequent status messages are sent after each request to modify Flash memory is received. The status returned is used to determine if the programming voltage was present, or whether or not the attempted modification was successful.

The flowchart in Figure 4 explains the operation of the Flash modifying routine, while Table 2 explains how the CAN protocol functions.

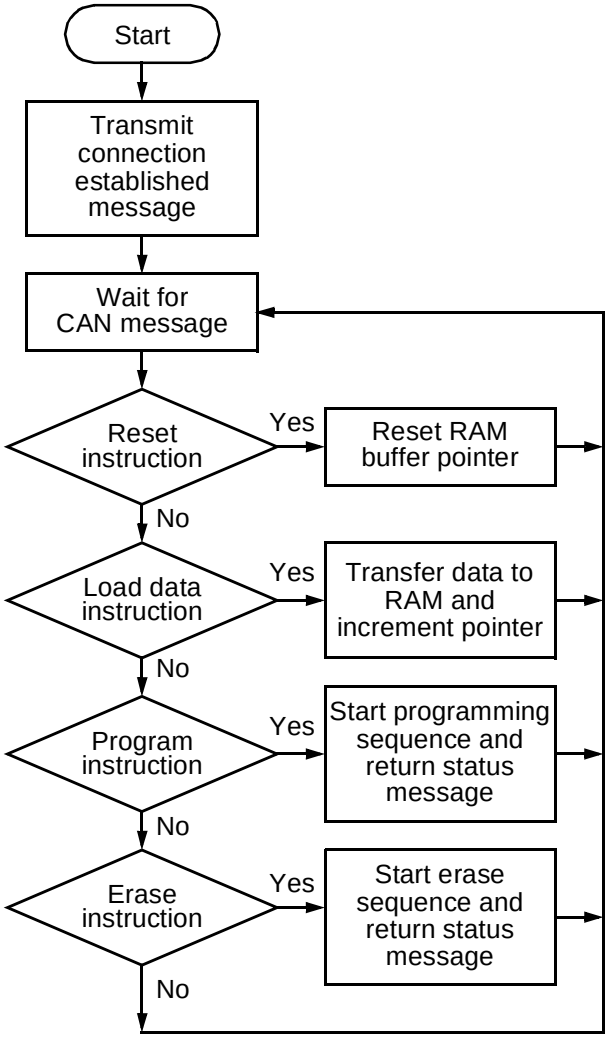

Figure 4. Flash modifying process flow

Table 2. CAN message Rx buffer contents

CAN Rx Buffer	Reset Instruction	Load Data Instruction	Program Instruction	Erase Instruction
DSR0	0	2	4	6
DSR1	—	Data Byte 1	Address MSB	Address MSB
DSR2	—	(Data Byte 2)	Address LSB	Address LSB
DSR3	—	(Data Byte 3)	No of Bytes	Word MSB
DSR4	—	(Data Byte 4)	(Page No)	Word LSB
DSR5	—	(Data Byte 5)	—	—
DSR6	—	(Data Byte 6)	—	—
DSR7	—	(Data Byte 7)	—	—

The reset instruction initializes a pointer to the start of a RAM buffer, where subsequent data bytes are incrementally stored until a new reset instruction is received. The data instruction may contain up to seven bytes of data. On receiving the program instruction, an attempt is made to program the specified number of bytes from the start of the RAM buffer into Flash, starting at the address sent in the instruction. The page number is optional and is included to provide support for S2 Records (> 64K). The erase command contains the starting address and word size of the flash block that has to be erased, this is required in order to allow verification of the erase process. Both the program and erase instructions return a status message, which provides information on the outcome of any attempt to modify Flash memory, whilst also providing a mechanism for data flow control to the target ECM.

Listing 2 provides a coded example of how to implement the Flash modifying via CAN protocol shown in Figure 4 and Table 2.

Listing 2 Flash modifying via CAN routine

```
;*******************************************************************************
; A bootloader routine to program 1.5T Flash via CAN
; Written to run on an MC68HC912BC32
;*******************************************************************************

;*******************************************************************************
; Register definitions
;*******************************************************************************

COPCTL:         EQU    $16
TIOS:           EQU    $80

;*******************************************************************************
; Bit definitions for the TIOS register
;*******************************************************************************
IOS7:           EQU    $80
IOS6:           EQU    $40
IOS5:           EQU    $20
IOS4:           EQU    $10
IOS3:           EQU    $08
IOS2:           EQU    $04
IOS1:           EQU    $02
IOS0:           EQU    $01
;*******************************************************************************
TCNTH:          EQU    $84
TSCR:           EQU    $86

;*******************************************************************************
; Bit definitions for the TSCR register
;*******************************************************************************
TEN:            EQU    $80
TSWAI:          EQU    $40
TSBCK:          EQU    $20
TFFCA:          EQU    $10
;*******************************************************************************

TMSK2:          EQU    $8D

;*******************************************************************************
; Bit definitions for the TMSK2 register
;*******************************************************************************
TOI:            EQU    $80
PUPT:           EQU    $20
RDPT:           EQU    $10
TCRE:           EQU    $08
PR2:            EQU    $04
```

```
PR1:            EQU    $02
PR0:            EQU    $01
;*******************************************************************************

TFLG1:          EQU    $8E

;*******************************************************************************
; Bit definitions for the TFLG1 register
;*******************************************************************************
C7F:            EQU    $80
C6F:            EQU    $40
C5F:            EQU    $20
C4F:            EQU    $10
C3F:            EQU    $08
C2F:            EQU    $04
C1F:            EQU    $02
C0F:            EQU    $01
;*******************************************************************************

TC0H:           EQU    $90
FEEMCR:         EQU    $F5

;*******************************************************************************
; Bit definitions for the FEEMCR register
;*******************************************************************************
BOOTP:          EQU    $01
;*******************************************************************************

FEECTL:         EQU    $F7

;*******************************************************************************
; Bit definitions for the FEECTL register
;*******************************************************************************
FEESWAI:        EQU    $10
SVFP:           EQU    $08
ERAS:           EQU    $04
LAT:            EQU    $02
ENPE:           EQU    $01
;*******************************************************************************

CMCR0:          EQU    $0100
```

```
;*******************************************************************************
; Bit definitions for the CMCR0 register
;*******************************************************************************
CSWAI:          EQU     $20
SYNCH:          EQU     $10
TLNKEN:         EQU     $08
SLPAK:          EQU     $04
SLPRQ:          EQU     $02
SFTRES:         EQU     $01
;*******************************************************************************

CBTR0:          EQU     $0102
CRFLG:          EQU     $0104
;*******************************************************************************
; Bit definitions for the CRFLG register
;*******************************************************************************
WUPIF:          EQU     $80
RWRNIF:         EQU     $40
TWRNIF:         EQU     $20
RERRIF:         EQU     $10
TERRIF:         EQU     $08
BOFFIF:         EQU     $04
OVRIF:          EQU     $02
RXF:            EQU     $01
;*******************************************************************************

CTFLG:          EQU     $0106

;*******************************************************************************
; Bit definitions for the CTFLG register
;*******************************************************************************
ABTAK2:         EQU     $40
ABTAK1:         EQU     $20
ABTAK0:         EQU     $10
TXE2:           EQU     $04
TXE1:           EQU     $02
TXE0:           EQU     $01
;*******************************************************************************

CTCR:           EQU     $0107
CIDMR0:         EQU     $0114
CIDMR2:         EQU     $0116

RXDSR0:         EQU     $0144
RXDSR1:         EQU     $0145
RXDSR3:         EQU     $0147
RXDLR:          EQU     $014C
```

```
TX0IDR0:        EQU     $0150
TX0DSR0:        EQU     $0154
TX0DLR:         EQU     $015C
TX0PRI:         EQU     $015D

;*****************************************************************************
; Standard equates
;*****************************************************************************

StackTop:       EQU     $0BFF
StartOfRAM:     EQU     $0800

CBT:            EQU     $C749
ConnectedMsg:   EQU     $55
LastInstr:      EQU     $06

EClock:         EQU     8000000           ;E-clock frequency in Hz

PrescaleBy32:   EQU     5                 ;generate msec delays based on 8MHz bus
ms10:           EQU     EClock/3200
ms1:            EQU     EClock/32000

PrescaleBy1:    EQU     0                 ;generate usec delays based on 8MHz bus
us22:           EQU     ((EClock/10000)*22)/100
us11:           EQU     ((EClock/10000)*11)/100

MaxProgPulses:  EQU     50
MaxErasePulses: EQU     5

                ORG     StartOfRAM

;*****************************************************************************
; Declare variables
;*****************************************************************************

pulseTotal:     DS.B    1                 ;tracks program/erase pulses applied
marginFlag:     DS.B    1                 ;indicates if prog or margin pulses
bytesTotal:     DS.B    1                 ;number of bytes to be programmed
```

```
;*******************************************************************************
; Initialization routine
;*******************************************************************************

bootloader:
                LDS     #StackTop          ;initialize stack pointer
                CLR     COPCTL             ;switch off COP watchdog
                BSET    TSCR,#(TEN+TFFCA) ;enable timer, allow fast flag clears
                BSET    TIOS,#IOS0         ;set channel 0 to output compare
                BCLR    FEEMCR,#BOOTP      ;enable erasure of protected block

;*******************************************************************************
; Setup the CAN module
;*******************************************************************************

                BSET    CMCR0,#SFTRES      ;place CAN module in reset
                MOVW    #CBT,CBTR0         ;set up CAN bit timing

                CLR     CTCR
                MOVW    #$FFFF,CIDMR0
                MOVW    #$FFFF,CIDMR2      ;set up module to receive all messages

                MOVW    #$0000,TX0IDR0     ;stanard ID (0 value)
                MOVB    #$01,TX0DLR        ;single byte status message
                CLR     TX0PRI             ;set status message registers

                BCLR    CMCR0,#SFTRES      ;take CAN module out of reset
canSynch:
                BRCLR   CMCR0,#SYNCH,*     ;synchronize module with CAN bus

                MOVB    #ConnectedMsg,TX0DSR0
                JSR     canTx              ;transmit connected status message
;*******************************************************************************
; Wait for CAN message
;*******************************************************************************

waitForMsg:
                BRCLR   CRFLG,#RXF,*       ;wait for CAN message

                BRSET   RXDSR0,#$01,waitForNextMsg
                LDAB    RXDSR0
                CMPB    #LastInstr
                BHI     waitForNextMsg     ;ignore invalid instructions

                CLRA                       ;jump to appropriate routine, depending
                JMP     [D,PC]             ;on instruction value
                DC.W    setupBuffer        ;(0) reset RAM ptr
```

```
                DC.W    loadBuffer          ;(2) load buffer up to a max 256 bytes
                DC.W    programFlash        ;(4) program flash with buffer contents
                DC.W    eraseFlash          ;(6) erase flash array

;*******************************************************************************
; Setup buffer pointer
;*******************************************************************************

setupBuffer:
                LDX     #ramBuffer
                BRA     waitForNextMsg

;*******************************************************************************
; Transfer data into buffer
;*******************************************************************************

loadBuffer:
                LDAB    RXDLR               ;number of bytes transmitted
                LDY     #RXDSR1             ;start of transmitted data
nextDataByte:
                DECB                        ;ignore command byte
                BEQ     endOfData           ;stop at end of data
                MOVB    1,Y+,1,X+           ;store data in RAM buffer
                BRA     nextDataByte
endOfData:
                BRA waitForNextMsg

;*******************************************************************************
; Program ram buffer contents into flash memory
;*******************************************************************************

programFlash:
                BSR     beginProgramming
                JSR     canTx               ;transmit status/flow control message
                BRA     waitForNextMsg

;*******************************************************************************
; Erase flash memory block
;*******************************************************************************

eraseFlash:
                JSR     beginErasing
                JSR     canTx               ;transimit status/flow control message
```

```
;*******************************************************************************
; Clear CAN Rx flag and wait for next message
;*******************************************************************************

waitForNextMsg:
                MOVB    #RXF,CRFLG          ;clear Rx flag
                BRA     waitForMsg          ;normal operation

;*******************************************************************************
; Flash programming algorithm
;*******************************************************************************

beginProgramming:
                LDX     #ramBuffer          ;point to the start of the RAM buffer
                LDY     RXDSR1              ;point at location(s) to be programmed

                CLR     TX0DSR0             ;indicates result of program procedure

                LDAB    RXDSR3              ;number of bytes to be programmed
                BEQ     progStatus          ;check for zero bytes to be programmed
                STAB    bytesTotal          ;store number of bytes to program
                BRCLR   FEECTL,#SVFP,progStatus
                                            ;check Vfp level
                INC     TX0DSR0             ;(1) indicate that Vfp is present
                MOVB    #PrescaleBy1,TMSK2
                                            ;setup timer prescalar for usec delays
progNextLocation:
                CLR     pulseTotal
                CLR     marginFlag          ;reset pulse total and margin flag

                BCLR    FEECTL,#ERAS        ;configure flash array for programming
                BSET    FEECTL,#LAT         ;enable addr/data latches
                MOVB    ,X,,Y               ;write data to flash address
progPulseLoop:
                BSET    FEECTL,#ENPE        ;switch on Vfp onto array

                LDD     #us22               ;generate 22 usec delay
                ADDD    TCNTH
                STD     TC0H
                BRCLR   TFLG1,#C0F,*

                BCLR    FEECTL,#ENPE        ;switch off Vfp from array

                LDD     #us11               ;generate 11 usec delay
                ADDD    TCNTH
                STD     TC0H
                BRCLR   TFLG1,#C0F,*
```

```
                TST     marginFlag          ;are margin pulses being applied ?
                BNE     progMargin

                INC     pulseTotal          ;update pulse count
                BRA     progCheck
progMargin:
                DEC     pulseTotal          ;apply the same number of margin pulses
                BNE     progPulseLoop       ;as there were programming pulses
progCheck:
                LDAB    ,Y                  ;read location being programmed
                CMPB    ,X                  ;compare against intended value
                BNE     progFail

                TST     pulseTotal          ;if 0 then margin pulses have been done
                BEQ     progSuccess         ;byte has been programmed

                INC     marginFlag          ;set margin flag if byte programmed
                BRA     progPulseLoop       ;and apply margin pulses
progFail:
                LDAA    pulseTotal          ;if 0 then margin pulses have been done
                BEQ     progStatus          ;and program has failed, TX0IDR = 1

                CMPA    #MaxProgPulses      ;if max program pulses have been applied
                BEQ     progStatus          ;no need to apply margin pulses

                BRA     progPulseLoop       ;continue applying program pulses
progSuccess:
                CLR     FEECTL              ;release LAT bit
                INX                         ;point to next data byte
                INY                         ;point to next flash location
                DEC     bytesTotal
                BNE     progNextLocation    ;program all bytes

                INC     TX0DSR0             ;(2) program successful
progStatus:
                CLR     FEECTL              ;release LAT bit
                RTS
```

```
;*******************************************************************************
; Flash erase algorithm
;*******************************************************************************

beginErasing:
                LDX     RXDSR1            ;(DSR1:2) start address of array
                                          ;(DSR3:4) size of array (in words)
                CLR     TX0DSR0           ;indicates result of erase procedure
                BRCLR   FEECTL,#SVFP,eraseStatus
                                          ;check Vfp level
                INC     TX0DSR0           ;(1) indicate that Vfp is present
                MOVB    #PrescaleBy32,TMSK2
                                          ;setup timer prescalar for msec delays
                CLR     pulseTotal
                CLR     marginFlag        ;reset pulse total and margin flag

                BSET    FEECTL,#(LAT+ERAS)
                                          ;enable addr/data latches and erase bit
                STAA    ,X                ;write to valid location in array
erasePulseLoop:
                BSET    FEECTL,#ENPE      ;switch on Vfp onto array

                LDD     #ms10             ;generate 10 msec delay
                ADDD    TCNTH
                STD     TC0H
                BRCLR   TFLG1,#C0F,*

                BCLR    FEECTL,#ENPE      ;switch off Vfp from array

                LDD     #ms1              ;generate 1 msec delay
                ADDD    TCNTH
                STD     TC0H
                BRCLR   TFLG1,#C0F,*

                TST     marginFlag        ;are margin pulses being applied
                BNE     eraseMargin

                INC     pulseTotal        ;update pulse count
                BRA     eraseCheck
eraseMargin:
                DEC     pulseTotal        ;apply the same number of margin pulses
                BNE     erasePulseLoop    ;as there were programming pulses
eraseCheck:
                LDX     RXDSR1            ;start of array
                LDY     RXDSR3            ;word size of array
                LDD     #$FFFF            ;erased state of word
```

```
eraseCheckLoop:
                CPD     2,X+                ;check all array entries are erased
                BNE     eraseFail
                DBNE    Y,eraseCheckLoop

                TST     pulseTotal          ;if 0 then margin pulses have been done
                BEQ     eraseSuccess        ;array has been erased

                INC     marginFlag          ;set margin flag if array erased
                BRA     erasePulseLoop      ;and apply margin pulses
eraseFail:
                LDAA    pulseTotal          ;if 0 then margin pulses have been done
                BEQ     eraseStatus         ;and erase has failed, TX0IDR = 1

                CMPA    #MaxErasePulses
                                            ;if max erase pulses have been applied
                BEQ     eraseStatus         ;no need to apply margin pulses

                BRA     erasePulseLoop      ;continue applying erase pulses
eraseSuccess:
                INC     TX0DSR0             ;(2) array is erased
eraseStatus:
                CLR     FEECTL              ;release LAT and ERAS bits
                RTS

;*****************************************************************************
; CAN status/flow control message transmit routine
;*****************************************************************************

canTx:
                BRCLR   CTFLG,#TXE0,*       ;wait until Tx buffer is available
                MOVB    #TXE0,CTFLG         ;transimit status/flow control message
                RTS

;*****************************************************************************
; Label pointing to first available ram location after bootloader for buffer
;*****************************************************************************

ramBuffer:

;*****************************************************************************
```

When considering potential improvements to this part of the system, it is worth noting that most will be technology and / or device dependent, although by extending the CAN protocol it should be possible to support most, if not all, of the potential enhancements. For instance, the CAN protocol could be modified to include support for paged memory devices, such as Motorola's MC68HC912DG128, as shown in Table 2. The protocol could also be used to provide access control to protected memory areas or extend the capabilities of the status messaging, e.g. return failing address information.

However, there are some extensions that need more than just a modification of the CAN protocol, for example, to include eeprom support would require the inclusion of device specific program and erase routines. In the case of the HC12 family, an external 12 volt programming voltage is required, but in order to limit the programming interface to just the CAN wires, additional hardware (i.e. 12 volt charge pump) must be included on the PCB. The charge pump itself could be enabled by one of the MCU output pins, which in turn could be controlled through the CAN protocol or directly from the Flash modifying algorithms. The HC08 family includes an onboard charge pump and as such does not require the inclusion of any additional hardware.

6 A ‘smart cable’ to interface between a PC and the target ECM

If you consider that the minimal requirement for this part of the system is to provide the hardware to convert an S-Record into a stream of CAN messages, suitable for use with the protocol described in Table 2, purchasing one of the many commercially available CAN PC interface cards is all that is required. However this approach requires much more effort during the software development of a suitable API, it is also less portable, with each PC or Laptop requiring the appropriate piece of hardware to be installed before it can be used as part of the system.

An alternative approach is to develop an additional piece of hardware that provides a CAN interface to the target ECM and either a serial or parallel interface to the PC or Laptop. A prototype system was built using the M68EVB912BC32, which has all the necessary hardware requirements, e.g. RS232 and CAN physical interfaces.

Having an MCU based 'smart cable' provides the ability to add a lot of additional functionality, meeting each system's individual requirements. The prototype system included a lot of extra features, which although not necessary enhanced the performance of the overall system.

Data flow control between the PC and the target ECM is essential and can be handled by the smart cable. System parameters, including baud rate, smart cable operating frequency, CAN bit timing, LRAE maintenance identifier and target MCU details are all stored in the cables non-volatile memory. Each of these parameters can be modified and, in case of error, restored to a set of default values, some changes do not take effect until the smart cable itself is reset.

Error handling can also be included, validating S-Records, verifying target device memory mapping, reporting system errors and failures during Flash modifying routines.

The flowchart in Figure 5 explains the operation of the prototype smart cable.

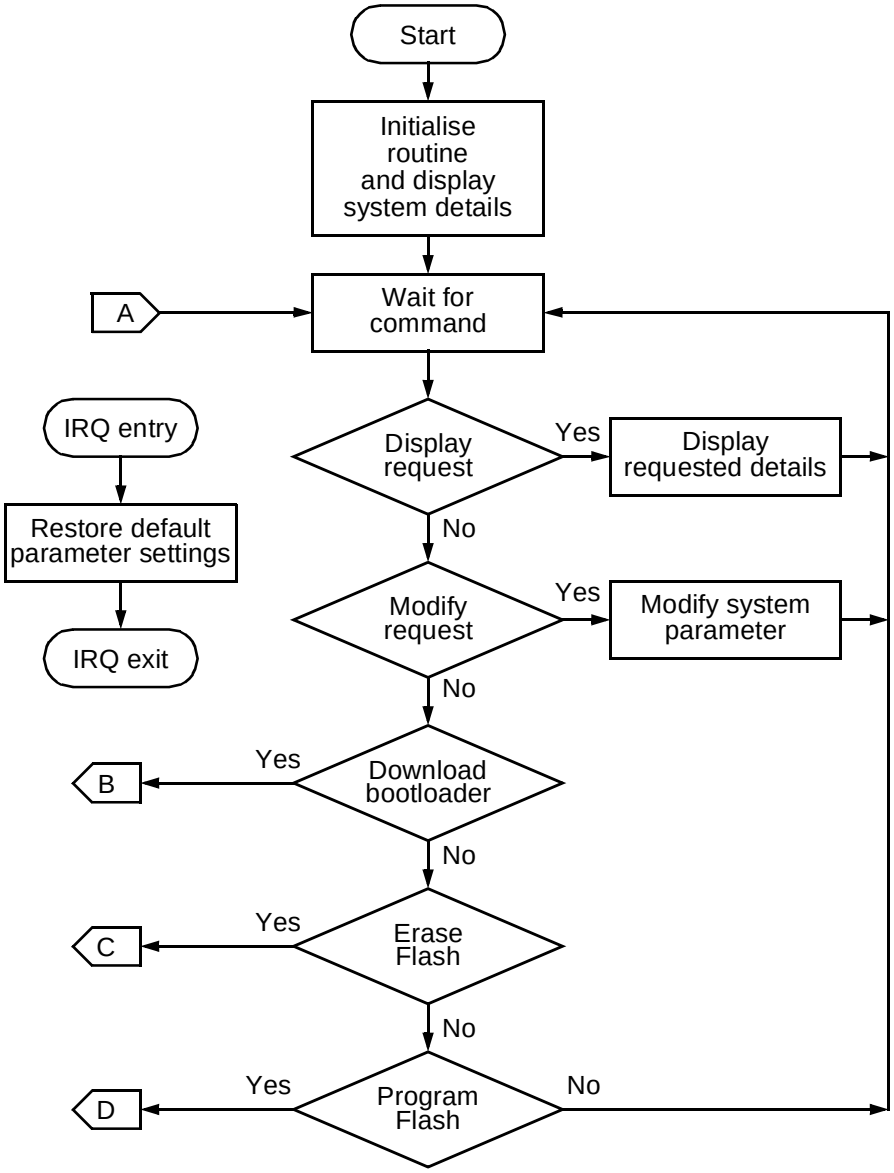

Figure 5. Smart cable process flow (Sheet 1 of 3)

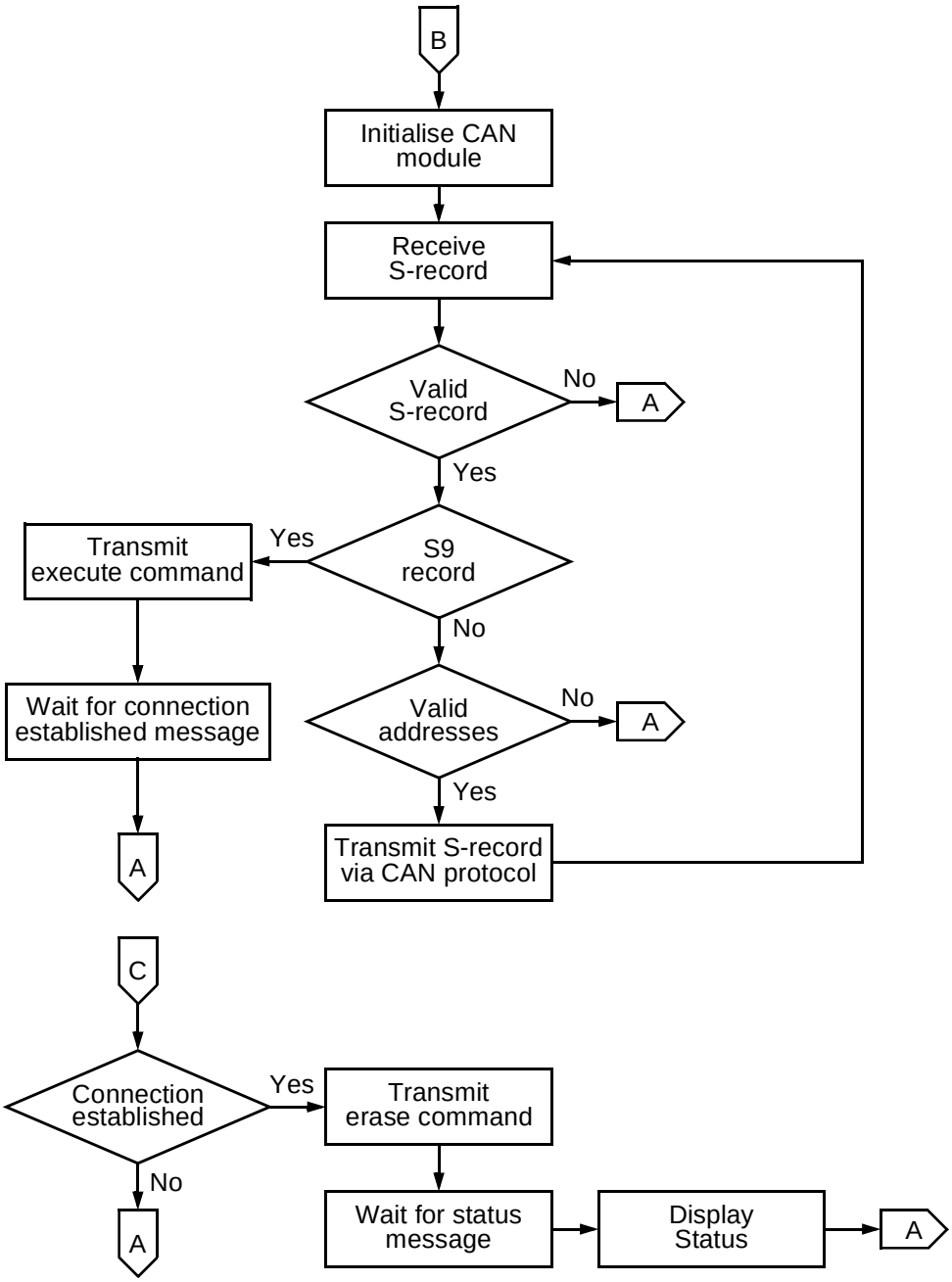

Figure 5. Smart cable process flow (Sheet 2 of 3)

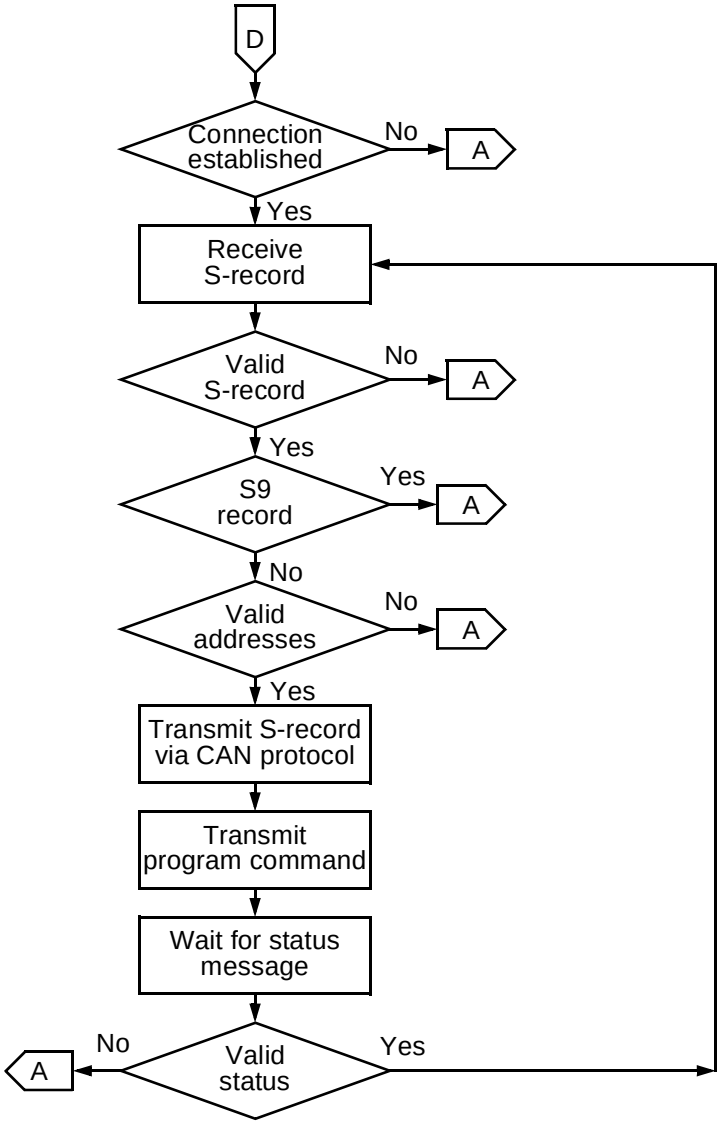

Figure 5. Smart cable process flow (Sheet 3 of 3)

The smart cable receives either commands or S-Records from the PC, each command is processed and the appropriate action taken. On receiving an S-Record, the cable validates and performs a range check, based on the target MCU specified, and if appropriate translates it into a suitable CAN format for communication to the target ECU.

The actual features included on the smart cable can be modified to suit the individual needs of each system. It could be used to supply the programming voltage, when appropriate, it could be optimized for speed (e.g. parallel communication), provide an additional level of security or include some diagnostic capabilities.

7 An API capable of transferring data to the ‘smart cable’

The actual requirements for this part of the system are dependent upon the approach taken in the previous section, by developing the 'smart cable' concept the minimal requirements for the API are greatly simplified.

The prototype smart cable was designed to accept either S-Records or ASCII text strings via a serial interface. To successfully process a stream of S-Records, the cable transmits a pace character to inform the API it is ready to receive new data.

A terminal emulator, that supports the pace character flow control method, provides all the necessary functionality required when using the prototype system.

Developing a custom API could be used to provide additional features, simplify the user interface or improve the overall look of the product.

8 Additional information

The Flash programming algorithm shown in section 5 utilizes a very simple data transfer scheme, it receives and stores data from a single S-Record into a ram buffer, the buffer data is then used to program the Flash array before a request for new data is issued. Using this method results in an overall operating time equal to total transmission time plus total programming time.

The following example provides a comparison of programming time versus transmission time and makes the following assumptions:

- Assume the command byte is encoded into the CAN ID, allowing transmission of up to 8 data bytes per CAN message
- CAN transmission at 125Kbits/s, using extended ID’s and ignoring bit stuffing
  - Buffer reset command (0 data bytes / 64 bits) takes 0.512 milliseconds
  - Load buffer command (8 data bytes / 128 bits) takes 1.024 milliseconds
  - Program buffer command (4 data bytes / 96 bits) takes 0.768 milliseconds

- Each S-Record contains 32 bytes of data requiring
  - One buffer reset command
  - Four load buffer commands
  - One program buffer command
- Require to program all 32,768 bytes of the MC68HC912BC32
  - 1024 S-Records needed in total

Therefore,

Total transmission time = 1,024 x (0.512 + (4 x 1.024) + 0.768) msecs
= 5.505 seconds

By assuming the average number of programming and margin pulses to be 5 in total, a programming pulse of 25 microseconds and a time to verify of 15 microseconds,

Total programming time = 32,768 x (5 x (25 + 15)) usecs
= 6.554 seconds

At first glance, an overall operating time of approximately 12 seconds might be considered acceptable, but if the target device is changed to an MC68HC912DG128, the total time jumps to nearly 50 seconds. Another consideration, although not part of the subject matter of this paper, is the possibility that another, slower, serial protocol could be used to transfer data, such as J1850 or a UART based system. With transmission rates dropping as low as 10 Kbits/s, suddenly transmission time becomes the biggest influence on the overall operating time.

The benefits to be gained by optimizing programming algorithms for speed vary from convenience, during the development cycle, to cost savings in a production environment. There are several techniques that can be used to reduce operating time, with varying degrees of success, i.e. using the CAN identifier to encode an additional three bytes of data reduces the number load buffer commands from 4096 to 2979, saving over a second in the previous example.

However the biggest return in time saving comes about through the adoption of a parallel programming algorithm, i.e. data is continually received into a circular buffer whilst programming is carried out simultaneously from the same buffer.

When employing this method care has to be taken to avoid an over run condition between the load and program operations, but in return the overall operating time should be limited to the larger of the two values, total transmission time or total programming time.

9 Conclusion

By including the CAN LRAE feature to a system specification, the potential benefits that can be gained outweighs the effort required in meeting that specification.

However, care should be taken when identifying the exact system requirements, as the LRAE function provides the backbone that defines each system's limitations.

It is also apparent that a great deal of flexibility exists when defining the additional tools required to support this application. By developing custom hardware it is possible to significantly reduce the work involved in producing a suitable API, whereas selecting a readily available piece of hardware increases the work required on the API. This flexibility enables designers to develop a system best suited to the available skill set at their disposal.

By designing the LRAE routine to be compatible with one of the existing CAN standards, such as CCP, it may be possible to purchase a commercially available product, capable of providing both the smart cable and API functionality. Although this approach only requires the development of the LRAE section, it restricts the amount of customization that could otherwise be achieved.

A basic implementation of the complete system is possible with surprisingly little effort, with the amount of additional work required dependent upon the level of customization undertaken to meet the overall system specification.

This Page Has Been Intentionally Left Blank

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

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

ASIA/PACIFIC:

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

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://www.motorola.com/semiconductors