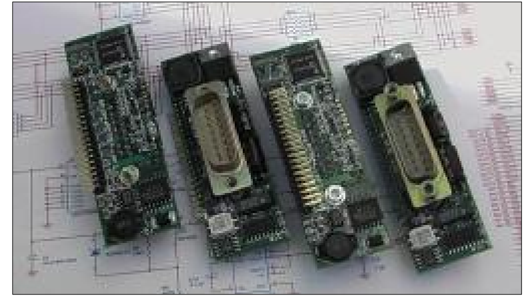


FIRMWARE

Internal Reference

CAN-DO!

CREATED BY STEPHEN M MORACO – KC0FTQ
SPONSORED BY AMSAT-DL

CAN-DO! Firmware Internal Reference

Document Created by:
Stephen M Moraco, September 2004
<kc0ftq@amsat.org>

Last updated: 7 June 2005

Table of Contents

Introduction 3
1: Logical Layout 4
  Order of elements 4
  Firmware Walk-through 6
    Standard and Multiplex Modes 8
    Byte-Pipe Mode 9
2: Physical Layout 11
  Build Tools 11
  Organization of the firmware source code 11
  What a build does 11
  Make targets 12
  How the files are laid out 12
  How to build the firmware 13
Appendix A: Revision History 14

List of Figures

Figure 1 - Build Flow 11

Introduction

This document presents information, normally deduced from wandering about a source tree studying the source code and Makefiles. It discusses the firmware organization and function from an internal perspective in the chapter entitled "Logical Layout" and the external organization of the flight source code in the chapter entitled "Physical Layout". As of this writing, there is no discussion of the various debug forms of source or the special test hardware source. However, they are very similar in layout, so understanding the flight firmware organization will serve well when trying to understand the others.

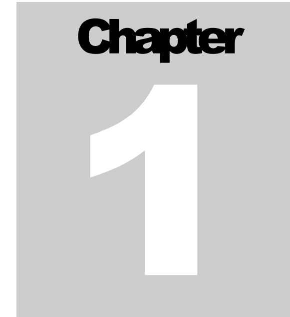
This is a companion document to:

1. The **CAN-DO! Users Guide**, which provides information about how to interact with a running widget and how to integrate a widget into a new module design.

2. The CAN-DO Widget **Schematic v1.6**

3. The ATMEL Document entitled "**Enhanced 8-bit Microcontroller with CAN Controller and Flash Memory – T89C51CC01**" available as a free download from the ATMEL web site.

Links to the CAN-DO documents and to the ATMEL web site can be found at the CAN-DO project website: http://can-do.moraco.info in the reference section.



1: Logical Layout

Releases of the source code are in the form of a single Intel Hex file, a single assembly listing file and a single layout report file (link map equivalent). This chapter explores the layout of this single assembly listing file in the section entitled "Order of elements" and it follows with a walkthrough of the firmware function in the section "Firmware Walkthrough".

Order of elements

The ROM (FLASH) in our ATMEL T89C51CC01CA part consists of 16 2KB pages numbered 0 thru F (15). The assembler does not manage this boundary automatically. Instead, we must decide where the code lives within these 16 pages and force the code to be in the desired page. In the flight firmware source file the content appears in the following order (the 2KB page is shown in parenthesis when it is set in the code).

- T89C51CC01 Register Equates
    - These equates describe the full set of registers, which are specific to this ATMEL part.
- Loop ID Codes
    - These are equates which describe which hard loop the firmware has entered. If the watchdog expires then the event recording system on restart will record the current loop id in the expiration event log.
- Register/RAM memory map
    - These are also equates which describe the purpose of each RAM byte and the registers.
- Reset Vectors and Startup Code (First real code is here) [Code Pg0]
    - This section starts with the hardware interrupt vectors. In some cases, the interrupt service routine is small enough to be contained in the vector. In the remaining cases, it is not so the vector contains some code and a jump to the remaining interrupt service routine code.

- Power-on Startup
    - After the vectors and interrupt service routines, the power-on code calls all of the device initialization routines and then finishes with a jump to main.
- CAN I/O Routines
    - Servicing the CAN Macro (peripheral in the ATMEL part) is broken up into a number of small subroutines. All of the subroutines directly interacting with the CAN controller reside in this section.
- Main [Code Pg1]
    - After initialization is complete, the code enters a message-processing loop. The activity within the loop is specific to the mode of device. For the Standard and Multiplex modes the processor enters IDLE (low-power) mode and the CAN Interrupt Service Routine awakens when a CAN message to arrives, processes it, generates one or more response messages and then exits. In the case of Byte-Pipe Mode, IDLE mode is not used, instead there is a state machine, which keeps the input and output data moving while processing the CAN message traffic. The diagram for this state-machine is available at the can-do project web site. (See: http://can-do.moraco.info/review/CAN-DO-pipeStates.htm)
    - The remainder of this code in this section is message-processing subroutines.
- General Purpose sub-routines (Utility Routines) [Code Pg3]
    - Simple utility routines, see the source code for detail.
- ADC I/O Routines
    - The ATMEL part contains an Analog to Digital Conversion (ADC) peripheral. There are eight inputs multiplexed to this ADC. Routines interacting with this ADC are in this section.
- Memory Checksum Routines
    - Routines performing any kind of checksum on any of the memory regions are contained in this section.
- Address/Mode Input Routines
    - The routines in this section provide access to the shift register, our only means of access to the Address and Mode settings (jumper pads).

- Pipe Watch Dog Routines
    - This section contains routines interacting timers, which are implemented using the Programmable Counter Array (PCA) assets. These routines implement the Byte Pipe Input and Output timer functionality.
- Hardware Watch Dog Routines
    - Routines in this section provide control over the Hardware Watch Dog mechanism.
- Serial I/O Routines
    - This section is a collection of subroutines useful when the serial port is active. In the three main modes, this code is not used. In special modes and during debug this code is used. For flight, this code simply sets the Serial Hardware to a disabled state.
- End ROM Equates, ROM Checksum word, etc. [tail-end of Code PgF]
    - Our FLASH image ends with final checksum values inserted by the flasher utility. The firmware sums all bytes of itself at power-on expecting to arrive at a zero sum. Equates in this section define the location where the ending checksum is stored.

Now we have seen the order of sections within the firmware assembly listing, let us look at the expected behavior of the firmware.

Firmware Walk-through

This walk-through is a lightweight overview of the firmware organization and its function. There is a large amount of inline documentation within the source code. If any conflict arises between this write-up and the source code, the source code supersedes. Finally, you will find address references within this narrative. These are accurate for version 1.10 only. They should still be nearby for later versions but prior to v1.08 the routines may be in very different locations. So, let us get on with the tour.

The CAN-DO firmware runs the ATMEL t89c51cc01ca microcontroller onboard the CAN-DO Widget. It supports three operating modes: Standard Mode, Multiplex-Mode and Byte-pipe mode. Shorting a combination of the two pads labeled M0 and M1 selects the desired mode.

Most of the analog reading routines and the CAN message reception and transmission routines are common to all three modes. The modes differ in how the configure (command) and response (telemetry) messages are handled.

At the front of the listing, we see equates for all the bits and bytes used in RAM and those, which are control registers. Skipping past all of this we locate the address 0x0000 (the equates have '=' after them, the addresses have a ':' so we are looking for '0000:') in our listing.

Immediately at the label 'reset:' which is found at address 0x0000, you'll see a long-jump to the main power-on routine. After this long-jump are the software interrupt vectors in addresses 0x0003 thru 0x004c which are followed by the two interrupt service routines: 'can_handler:' and 'pca_handler:'.

The 'poweron:' routine (overall device initialization) starts after these two interrupt handlers. The overall behavior of the 'poweron:' routine is to configure each of the onboard resources: RAM, ERAM, Shift Register, ADC, Timers, CAN Controller, Programmable Counter Array, H/W Watchdog, and Serial I/O. In addition to this basic setup, however, this 'poweron:' routine is responsible for detecting and recording the type of event that caused the current reset. If the reset cause was a Watchdog event and the RAM is still correct then it will also cause the last known configure values to be written to the digital outputs. The last act of the 'poweron:' routine is to then jump to the main loop at 'main:'.

At address 0x0800 is found the routine 'main:'. The first act when arriving at main: is to decide which main loop is appropriate for the mode selected by the M0, M1 jumpers. There are two different main loops: (1) 'std_main:' which is used for both the standard and multiplex modes and (2) 'pipe_main:' which is used only for the byte-pipe mode. Let us look at the 'std_mode:' version first.

The 'std_main:' routine enables the CAN and overall system interrupts and then enters a low-power IDLE state (halts). If some interrupt allows exit of the IDLE state the code loops back and will simply re-enable the interrupts and re-enter the IDLE state, forever. The CAN Interrupt Service Routine is intended to handle all the message traffic.

With the interrupts enabled and sitting in IDLE mode, everything now hinges on CAN messages arriving. The Widget effectively does nothing unless a CAN message arrives asking it to take some action. If it does not see a configure message in ~3.12 Seconds the hardware watchdog will expire resetting the processor and all starts over again from address 0x0000 (the reset vector).

When a CAN message arrives the 'CAN_handler:' interrupt service routine calls 'hndl_CANmsg:' (address 0x0815) which will determine which message has arrived and dispatches to the code to take the appropriate action. The structure of each of the routines handling arriving messages is similar. Let us look, for example, at what happens when a "Census Request" CAN message arrives, so we can understand this typical structure.

At 'hndl_CANmsg:' we see a list of tests for which CAN message arrived. If the message is a "Census Request" then the routine 'read_cenrqst:' unloads the message from the CAN receiver into our RAM and determines if there were defects in the message received. If during unload, the message was flawed, then the BADCANRX bit is set. Upon completion of the message unload the routine returns control and this bit is checked. If the message was bad (bad length [DLC] or critical error during receive, etc.) then the code simply calls 'clr_cenrqst:' to reset the CAN receiver and we have completed the handling for this flawed message.

If, instead, a good message was received the 'rply_cenrqst:' routine is called before the 'clr_cenrqst:' is called to reset things. The 'rply_cenrqst:' routine builds the response packet in RAM and then calls the transmit routine which uploads the packet to the CAN controller and tells it to transmit the message. After this transmit request, control returns to the code, which

then clears the "Census Request" receiver and returns to the interrupt service routine, which called it. Later in time, the message transmit completes at which time an interrupt is generated which will quiet the CAN transmitter mechanism.

Standard and Multiplex Modes

Now that we have seen the handling of a typical message, let us look at our most important message "Configure" to see what is happening during the processing of the message and during the preparation of the response messages.

There are three types of "Configure" messages, one for each mode. After the configure message is received and has been determined that it is a good message (not flawed) then one of the three routines: 'hndl_std_configure:', 'hndl_mux_configure:' and 'hndl_pipe_configure:' is executed. We will only look at the standard and multiplex versions of these routines since the pipe version is the same as the standard mode version.

The 'hndl_std_configure:' routine (address 0x08F2) sets the switched power based on the power control bit in the configure payload. It then creates the second in RAM copy of the packet for later use if a watchdog restart occurs. Next, it writes the values in the payload to the digital outputs, digitizes the analog inputs and reads the digital inputs. Now that it has all of the input data in RAM, it generates and sends the AN03 response message followed by the AN47 response message in turn. After sending both of these messages, the code resets the hardware watchdog, mechanism, re-enables the "configure" message receiver and exits the ISR allowing the processor to reenter the low-power IDLE mode until the next message is received.

The 'hndl_mux_configure:' routine (address 0x0921) performs the same steps as the standard mode version except in how it reads the digital inputs and writes the digital outputs. Instead of simply writing the configure payload values to the digital outputs an input/output multiplexer is implemented. This multiplexer writes each of eight bytes individually to the output port by first writing the latch address [0-7] then writing the byte of data, asserting the strobe, reading the input byte and finally, de-asserting the strobe. Likewise, in place of reading the digital inputs the multiplexer mechanism is driven again to re-read the digital inputs after all the time was spent digitizing the analog inputs. The act of reading the multiplexed digital inputs writes the bytes read into the payload RAM area. This payload is later sent as the IN07 reply. After this reply is sent both the AN03 and AN47 messages are built and sent as in the standard mode routine. In the multiplex mode, though, since the digital input data is in its own message (IN07) there are no digital input bits in the AN03 and AN47 messages.

Let us take a closer look at the analog digitizer routine common to all three modes. The routine 'precision_digitize:' (address 0x18B3) implements a 16x over-sample algorithm. It iterates over each of the eight analog channels [0-7], in order, accumulating the digitized sums for each channel. It makes 16 passes over all eight channels. After accumulating all the sample sums, it iterates over all eight dividing each by 16, which then produces the final value for each channel. The results are stored in RAM where the AN03 and AN47 build routines can access them at the time these messages are built and sent.

This completes the discussion of the two most common modes. Now let us look at the byte-pipe mode.

Byte-Pipe Mode

The alternate main routine is 'pipe_main:' (address 0x0B72). Conceptually, this routine is implementing the background task of a foreground and background pair. If one thinks of the foreground task as having the highest priority then the CAN ISR, which handles the CAN messages, represents the foreground task. Message reception and processing is fundamentally the same as in the standard and multiplex modes. Now things get different. In 'std_main:' we saw the processor simply go into an IDLE mode and effectively power down. In the byte-pipe mode, we instead implement a byte-pipe state machine, which is always running when the CAN ISR is not running.

The byte-pipe state machine is responsible for operating two independent transfer channels: IHU to Module (pipe-out) and Module to IHU (pipe-in). The pipe-out side waits for a received message (pipe-write) and then transfers each of the eight-bytes of the message to the digital outputs. This transfer occurs by writing the next byte; asserting the transfer request line (OUTREQ) and then waiting on the Module electronics to indicate that it received the output byte asserting the output acknowledge line (OUTACK). Before the output-side begins an eight-byte transfer it sets a timer to expire if this transfer of all eight bytes takes longer than normal. If this timer expires then the full transfer will not complete and the output state is reset to waiting for next (pipe-write) message. At the end of a successful eight-byte write to the module, the pipe-out timer is shutdown.

The pipe-in side is responsible for obtaining data from the module electronics and sending it as CAN messages (pipe-reads) to the IHU. The input-side state machine sits waiting for assertion of the input transfer request (INREQ) by the Module. If this happens the byte is read from the Digital Inputs and the transfer is acknowledged (INACK is asserted). Upon receipt of the first of each eight bytes an input side timer is started which will expire if the eight-byte input transfer takes too long. At the end receiving the eighth byte, the timer is halted and the pipe-read message is built and sent to the IHU. If, instead, the timer expired the bytes received so far are sent to the IHU, which means that a message with less than eight bytes is being sent in this case.

The input and output side state machines must work simultaneously so the final implementation of the state machine takes the form of six loops the code within each of which is headed by a block of "test and jump if case is met" instructions. Some of these tests are related to the timers expiring, some of these are related to checking the state of the appropriate transfer request and transfer acknowledge lines and the last group tests to see if a pipe abort was requested by the latest configure message received. Each test then jumps to code which implements the appropriate response to the event detected.

The pipe-mode AN03 message contains some state indicators of these two pipe-in and pipe-out state machines and some pipe event counters. With this state indication in the AN03 message and with control (reset ability) in the configure packet, the IHU software can, if desired, maintain knowledge of the health of the pipe-mode widget and can assert control over it when/if things go awry.

> *NOTE: for this reason we suggest each module designer when implementing pipe-mode electronics, use one of the remaining two digital output bits as a module electronics power-on reset if the designer wishes not to have the module power-cycled when/if the handshake gets confused between the module and the CAN-DO Widget.*

One final note on the differences between standard mode and byte-pipe mode: In order to implement byte pipe mode a pair of the analog input lines was used thus removing them from consideration/use by the module designer. This affects the implementation of the analog digitize routine which, in pipe-mode, skips channels AN0 and AN4 which are in this mode used as transfer control lines.

This overview does not mention many routines, as it is only an overview of function. There is much documentation in the source code, so please refer to it for more detail.

We have looked at the firmware source code and have a sense for its organization and function. The next chapter shows the division of source code amongst many files and identifies the tools and commands needed to build the code, the act of which produces the assembly listing file we have just studied.

Chapter
2

2: Physical Layout

This section describes the organization of the source code files and the purpose of miscellaneous files found in the firmware project directory. Build instructions are presented later in the section.

Build Tools

To build the firmware the as31 assembler executable must be present in its directory (sibling to the firmware directory in the source tree). When building the firmware both the as31 assembler and the mycpp preprocessor (found in the firmware directory) are used.

Organization of the firmware source code

Device specific routines and function specific routines within the firmware source code reside in different source files. Within the source files, conditional preprocessor statements enable selection of different functionality externally by the build mechanism (in this case, the commands in the Makefile.) These two mechanisms provide isolation of change to specific source files and allow the source to be multi-purposed without replication of code.

What a build does

By assembling, many source files {file}.a51 and included assembly source files {file}.i51, the final product an Intel Hex file ({target}.hex) is produced which is downloaded to the Widget (flashed). In order to get to this point the assembler source files necessary for a given target are combined into a single source file ({target}.cpp) which is then processed by mycpp producing a single conditional-statement-free source file ({target}.a51). This new single source file is then assembled by as31 producing both a listing file ({target.l51} and the desired hex file. Figure 1 depicts this process flow.

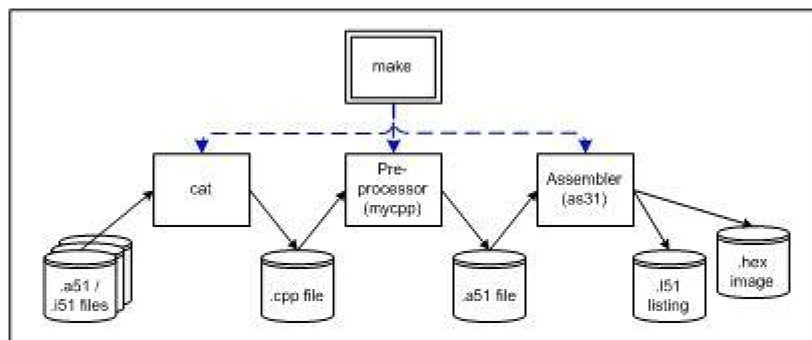


Figure 1 - Build Flow

Make targets

A number of different versions of the code are built (e.g., flight-ready, flight-code with debug or with serial port as debug, code for the PHYTEC development board, etc.) each of which produce .hex files. Some of these pertain to the Widget hardware while the rest are for the PHYTEC Development Board. This PHYTEC development board contains the exact same processor in a different physical package making it easy to test routines using it instead of the Widget board. At times, we use the PHYTEC board to emulate a module connected to a widget giving us the chance to exercise both sides of, say, a byte-pipe transfer using a single code base.

How the files are laid out

Files found in the firmware source tree belong to a couple different groups. There are files that describe the current state of the firmware (DESCR), files that comprise the core firmware (FLGIHT), those that add code for test hardware (PHYTEC, TEST), those that add descriptive output when the firmware is built for debug (FLT-DBG, PHY-DBG), and those that control build tools (BUILD) or are themselves build tools (TOOL). The following list identifies each file along with its purpose:

ChangeLog	DESCR: Running modification history for source code files
TODO.txt	DESCR: List of things needing attention or on wish list
adc.a51	FLIGHT: Routines providing interface to 8-channel Analog to Digital Converter (ADC) peripheral
canio.a51	FLIGHT: Routines providing interface to the CAN Controller peripheral
checksums.a51	FLIGHT: Routines for performing and validating checksums of memory regions
endmodule.a51	FLIGHT: Last file to be appended when creating single source file
loopcodes.i51	FLIGHT: Include file defining names for hardware loops
main-pipe.a51	FLIGHT: Main loop for pipe-mode operation
main-stdmux.a51	FLIGHT: Main loop for standard and multiplex mode operation
memory.a51	FLIGHT: Global Data definitions accounting for RAM, ERAM, FLASH and Register use
mem-utils.a51	FLIGHT: General purpose routines facilitating access to various memory types (ERAM, EEPROM, FLASH, etc.)
modinit.a51	FLIGHT: interrupt vectors, power-up and initialization code
pipe-wd.a51	FLIGHT: Byte-pipe timer routines
serial.a51	FLIGHT: Serial I/O Routines (used only while debugging)
shift-reg.a51	FLIGHT: Routines providing interface to address/mode Widget Jumpers
t89c51cc01.i51	FLIGHT: Include file defining full register set for T89C51CC01 part
utils.a51	FLIGHT: General utility routines making coding easier on 8051
wd.a51	FLIGHT: Routines providing control of Hardware Watchdog
canio-msgs.a51	FLT-DBG/PHY-DBG: Text messages used while debugging CAN interface routines

main-msgs.a51	FLT-DBG: Text messages used while debugging the firmware in general
phyinit.a51	PHYTEC: Main initialization code for PHYTEC development board (replaces modinit.a51)
phy-ui.a51	PHYTEC: user interface code for PHYTEC body of code (via Serial)
phymsgs.a51	PHY-DBG: Debug Messages for PHYTEC build of code
tioinit.a51	TEST: Main initialization code for Test Input/Output build of code (replaces modinit.a51)
Makefile	BUILD: Make rules
flight.ctl	BUILD: Disassembler control file (input to d51)
idcutset	TOOL: Perl Script: read .l51 file and produce link-map (.r51 file) and call/jump optimization list.
myUtils.pm	TOOL: Perl Module: subroutine library used by mycpp (cpp replacement)
mycpp	TOOL: Perl Script: custom replacement for cpp doing only what we need for this project

How to build the firmware

Make controls the build of the source code. The make utility follows instructions contained within the Makefile. The Makefile supports a number of targets (things to be built). To build any one of them you invoke make with the desired target name as the parameter. The following list shows the more useful make targets. Read the Makefile for the full latest list of supported targets.

Make full	Removes all intermediate files, then rebuilds all known .hex targets (synonym for 'Make clean all debug serial'
Make all	Makes sure all non-debug targets are current, rebuilding any which are not (this is the default make target executed when the command 'Make', with no parameters, is used.)
Make rpt	Make new versions of all non-debug .r51 report files if they are not already up to date.
Make fullrpt	Same as 'Make full' but also generates all of the .r51 files as well.
Make debug	Makes sure all 'dbg' targets are current, rebuilding any which are not
Make serial	Makes sure all 'sio' targets are current, rebuilding any which are not
Make {target}.hex	Make sure a specific .hex file is up-to-date. (The possible targets are flight-dbg.hex, testio-dbg.hex, flight-siodbg.hex, testio-siodbg.hex, flight.hex, testio.hex, and phyrad.hex)
Make clean	Removes all intermediate files leaving the .hex files for later download

Appendix A: Revision History

05 June 2005

First full release.

28 February 2005

First partial not-quite-a-release, some parts not yet present.

12 September 2004

Document Created