CAN Module Documentation

Thomas Craig – twc22

12/11/2009

Overview

Purpose

To provide a standard and robust C-language ARM7 software interface to the Controller Area Network (CAN) busses that form the main interconnect of Ranger's peripheral nervous system.

Problem Statement

Ranger's electronic nervous system essentially consists of two parts, the central and peripheral nervous systems. The peripheral nervous system in turn consists of a number of ARM7 microcontrollers boards, known as satellites, each connected to one or more CAN busses which each terminate at a central ARM7 known as the CAN Router. The CAN Router, an ARM9 microcontroller known as the Main Brain, and a high-speed Serial Peripheral Interface (SPI) bus interconnect between the two, together constitute Ranger's central nervous system.

Given the multitude of ARM7 microcontrollers accessing CAN busses, it is very desirable to have common software shared among all of the satellites for doing so. The main ostensible benefits of such a system are only having to debug the CAN software once and allowing programmers to add new boards and messages to the CAN busses with minimal effort.

The conceived requirements for the CAN Module were:

1) Support transferring frames with various mixed data-type payloads.
2) Automatically disseminate data immediately upon receipt.
3) Assemble data from remote locations into complete frames for transmission on demand.
4) Cleanly integrate with the task scheduler in order to schedule CAN transmissions.

Description of CAN Bus

The CAN Bus is a differential two-wire serial data bus that nominally operates at a raw bit rate of up to 1 MHz, although that has been successfully overclocked to 4 MHz on Ranger. The CAN bus works by transferring frames with payloads of up to eight bytes each. Each frame is transmitted with an identifier, known as the CAN ID, of either 11 bits in standard mode or 29 bits in extended mode. CAN

controllers do a substantial amount of work in hardware, including error detection and retransmission, multiple transmitter time-sharing and prioritization based on CAN ID, and frame parsing and filtering.

CAN Module Description

The CAN Module consists of two communicating layers, a frame transfer layer and a frame assembly and distribution layer. The purpose of the frame transfer layer is to send and receive complete frames of data over multiple physical CAN busses, while the purpose of the frame assembly and distribution layer is to collect data from disparate locations on a satellite into complete frames and to disseminate data from complete frames to disparate locations on satellites.

Code Structure

The evolution of the CAN module led to the intermixing of the source code for what became the two layers of module. The code is divided as follows:

- can.h: Shared header file for all parts of the CAN module.
- can_ring.c: Implements a reusable variable size ring buffer of `CAN_FRAME` struct elements.
- can_tx.c: Implements functions related to transmitting and assembling CAN frames.
- can_rx.c: Implements functions related to receiving and disseminating CAN frames.
- can_isr.c: Implements the interrupt service routines used by the CAN module.
- can_types.c: Implements functions related to the various supported frame layouts.

CAN Frame Ring Buffer

Concept

There is often need within the CAN module and those it interacts with for reusable ring buffer code for storing CAN frames. A ring buffer uses a contiguous block of memory, commonly known as an array, as its underlying storage area. However, unlike an array, the storage elements of a ring buffer are seen to be arranged in a ring, with no logical ends. This is similar to a queue data structure, but with fixed size. Unlike an array, a ring buffer must keep track of two indices, and input index and an output index. The input index points to the location in the ring containing the newest data, and the output index points to the location of the oldest data.

Implementation

Initialization

The `CAN_RING` struct must be instantiated and an array of `CAN_FRAME`s of the desired length must be allocated to serve as the underlying storage field. This `CAN_RING` instance is then initialized with the following function:

```
void can_ring_init(CAN_RING * ring, CAN_FRAME * frame_buf, int buf_len);
```

The first argument is the address of the ring to initialize, the second is the address of the buffer to use, and the third is the length of the buffer.

This initializes this input and output indices, `in_idx` and `out_idx`, to a value of `buf_len - 1`. To be specific, `in_idx` is defined as the index of the most recently input element, and `out_idx` is defined as the index of the most recently output element. Therefore, `in_idx` points to a valid data element when the buffer is not empty, but `out_idx` never points to a valid data element.

```
typedef struct can_ring{
  CAN_FRAME   *  buf;
  int            buf_len;
  volatile int   in_idx;
  volatile int   out_idx;
} CAN_RING;
```

```
typedef struct can_frame{
  CAN_CHANNEL   chan;
  int           addr : 11;
  int           dlc  :  4;
  char          rtr  :  1;
  CAN_PAYLOAD   payload;
} CAN_FRAME;
```

Push

Inserting an element into a CAN_RING is done through the `can_ring_push` function:

```
int  can_ring_push(CAN_RING * ring, CAN_FRAME * frame);
```

This function checks if there is room to add the given frame and returns 1 if there is no free space remaining in the ring. Otherwise, `in_idx` is incremented and rotated if necessary, the frame is inserted into the ring at that location, and 0 is returned.

Pop

Removing an element from a CAN_RING is done through the `can_ring_pop` function:

```
int  can_ring_pop(CAN_RING * ring, CAN_FRAME * frame);
```

This function checks if there is a frame available in the ring and returns 1 if the ring is empty. Otherwise, `out_idx` incremented and rotated, the next available frame is copied from the ring at that location to the frame location given as the second function argument, and 0 is returned.

Concurrency and Preemption

Although the prior sections glossed over it, the CAN_RING is designed to be safe to use between different preemption levels. Specifically, the CAN_RING is designed to such that it is always safe for the input end to be at a different preemption level than the output. However, it is not safe for a single end to be accessed by multiple preemption levels. Worded another way, operations must be atomic relative to other operations of the same type.

For example, this means that it is safe to connect a ring between main and interrupt, or between interrupt and fast interrupt levels, but it is not safe to push frames into a single ring from both main and interrupt levels.

Transfer Layer

Concept

The transfer layer is divided into two separate paths, the transmit path and the receive path. To transmit a frame, the user submits a frame to the transfer layer and it is adding to the transmit buffer for the appropriate CAN bus. Asynchronous transmit processes for each bus empty these buffers onto the wire.

The user does not directly interact with the transfer layer in order to receive a frame. Each channel can be configured to optional store received frames into a ring buffer or to automatically dispatch frames via the frame distribution layer.

Most functions within the transfer layer are written generically so as to apply to any CAN channel. This works well because all of the CAN controllers on the ARM7 processor are identical with the exception of the base address of their registers.

Implementation

Initialization and Channel Configuration

Since the transfer layer's code handles the CAN controllers in a generic manner, it was necessary to introduce structures to store the configuration and state of each channel. The `CAN_RX_CHAN_CFG` and `CAN_TX_CHAN_CFG` structures serve this purpose for the receive and transmit directions, respectively. It was convenient to divide these into two separate structures because the transmit and receive code which contain the arrays of the instances of these structures are in separate source files, `can_tx.c` and `can_rx.c`, respectively.

```
typedef struct can_tx_chan_cfg{
  volatile unsigned long    * base_addr;
  CAN_RING                  * ring;
  int                         stalled;
} CAN_TX_CHAN_CFG;

typedef struct can_rx_chan_cfg{
  volatile unsigned long    * base_addr;
  CAN_RING                  * ring;
  CAN_FRAME_DESC           ** descriptors;
  CAN_DISPATCH_MODE           dispatch_mode;
} CAN_RX_CHAN_CFG;
```

```
typedef enum can_dispatch_modes{
  CAN_DISPATCH_AUTO,
  CAN_DISPATCH_MANUAL
} CAN_DISPATCH_MODE;

typedef enum can_channels {
  CHAN_SSP  = 0,
  CHAN_CAN1 = 1,
  CHAN_CAN2 = 2,
  CHAN_CAN3 = 3,
  CHAN_CAN4 = 4
} CAN_CHANNEL;
```

For initializing the receive configuration structure, there is the function `can_rx_set_chan_cfg`:

```
void   can_rx_set_chan_cfg(CAN_CHANNEL chan,volatile unsigned long *
base_addr, CAN_RING * rx_ring, CAN_DISPATCH_MODE mode);
```

The arguments to this function are which channel to configure, the base address of the registers for the hardware CAN controller for that channel, a pointer to a ring for storing frames in manual dispatch mode, and flag selecting between manual dispatch mode (`CAN_DISPATCH_MANUAL`) or automatic dispatch mode via the distribution layer (`CAN_DISPATCH_AUTO`). Note that in automatic dispatch mode the ring is unnecessary and can be omitted by replacing it with 0, the null pointer. The receive configuration structure also has another field called `descriptors`. This field is related to the distribution layer and will be described there.

For initializing the transmit configuration structure, there is the function `can_tx_set_chan_cfg`:

```
void   can_tx_set_chan_cfg(CAN_CHANNEL chan,volatile unsigned long *
base_addr, CAN_RING * tx_ring);
```

This is essentially identical to its receive counterpart, except the ring is mandatory in all cases unless transmit functionality is not desired. The integer `stalled` flag will be described later in the Transmit Path section of the Transfer Layer documentation.

Interrupt Service Routines

There are a total of nine interrupt service routines (ISRs) in the transfer layer. Each of the four CAN controllers has one transmit ISR and one receive ISR, and there is one common error handling ISR.

Notably, the receive and transmit ISRs do not follow the convention of having a single instance of generic code that applies to all CAN controllers. However, this was necessary due to the behavior of the ARM7's Vectored Interrupt Controller (VIC). Specifically, upon firing of a vectored interrupt, the VIC looks up the programmed ISR address for that interrupt and calls that function, but gives no other direct indication of what the source of the interrupt was. Therefore, if a single ISR is shared among all controllers then that ISR must manually look up the source of the interrupt, which would be a slow process. However, by having a separate ISR for every interrupt, which ISR is called implies the source of the interrupt, and so therefore no source lookup is required. In order to prevent problems and

complications due to code duplication, only the bare minimum of required code is in the transmit and receive ISRs. Instead, they pass off control to generic functions to do the actual work, indicating which channel should be used.

The receive ISRs are named `can_rx1_isr`, `can_rx2_isr`, `can_rx3_isr`, and `can_rx4_isr`, with prototypes as follows:

```
__irq void can_rx1_isr(void);
```

The receive ISR is fired whenever a CAN controller receives a frame. Control is passed on to the function `can_rx_now` to continue generic processing.

Similarly, the transmit ISRs are named `can_tx1_isr`, `can_tx2_isr`, `can_tx3_isr`, and `can_tx4_isr`, with prototypes as follows:

```
__irq void can_tx1_isr(void);
```

The transmit ISR is fired whenever a CAN controller finishes transmitting a frame. Control is passed on to the function `can_tx_send_next_frame` to continue generic processing.

The CAN controller is capable of encountering a number of error states. Practically, the only error of concern is the transmit error counter limit. Whenever the bus encounters a transmit error, it increments the transmit error counter, and upon success it is decremented. When the transmit error counter reaches its limit of 255, the CAN controller is prohibited from transmitting frames until the error is explicitly cleared. To do this, there is the CAN error ISR which is shared by all CAN channels, `can_error_isr`:

```
__irq void can_error_isr(void);
```

The error ISR is fired whenever a CAN error occurs. The ISR then checks each channel to see if it is in a bus-off state. If it is, the bus is reset.

While it is possible for transmit errors to occur in normal operation, they are very unlikely to accumulate sufficiently to reach the error limit. However, transmit errors are extremely common during the development process when microcontrollers are being programmed, inserted, and removed from the network, and so automatic error recovery is therefore essential to an efficient development cycle.

Receive Path

As mentioned earlier, the very first event in the receive path is the firing of the CAN channel's interrupt service routine. This ISR does no work of its own and immediately passes control to the function `can_rx_now`, passing the CAN channel as a function argument.

```
void can_rx_now(CAN_CHANNEL chan);
```

This function then collects the components of the received frame's data from the CAN controller's registers and stores it into a `CAN_FRAME` structure instance. Once this frame has been assembled, the

controller is told to release the data so that it can receive another frame. Then, if the transmit layer was configured in manual dispatch mode during the initialization step, the frame is pushed into the receive ring buffer; the frame is lost if the ring is full. Otherwise, if automatic dispatch mode is in use, the frame is passed to the first function of the distribution layer, `can_rx_dispatch_frame`. If desired, users can check the `chan` field of the frame structure in order to determine which channel a frame was received on.

Note that this function, like most generic CAN functions, must do a small amount of work to access CAN registers. Specifically, the absolute address of the desired register must be computed based on the base address of the CAN controller in use. To ease this process, a few macros were defined.

First, a list of the relative offsets of all registers in a CAN controller was defined as follows:

```
#define CAN_MOD   (0x00)
#define CAN_CMR   (0x04)
#define CAN_GSR   (0x08)
…
```

Then, a macro is defined which computes and correctly casts the address of a register based on a base address and a relative offset as follows:

```
#define CAN_REG(base,offset) \
        (*((volatile unsigned long *) (((volatile unsigned char *)base) +\
offset)))
```

After setting this up, registers can be read and written simply as follows:

```
frame.addr             = CAN_REG(base,CAN_RID);
CAN_REG(base,CAN_CMR)  = 1<<2;
```

Transmit Path

The user initiates the process of transmitting a frame by calling the function `can_transmit_frame`.

```
int can_transmit_frame(CAN_FRAME * frame);
```

This function takes the given frame, determines which channel it should be transmitted on based on its `chan` field, and pushes it into the transmit ring buffer for that channel. As mentioned earlier, an asynchronous process empties this ring onto the CAN bus. Associated with this process is the `stalled` flag stored in the transmit configuration structure. This flag indicates whether or not the process is currently running or if it has stalled out because it ran out of data to transmit. After pushing the new frame onto the ring, this function checks the stall flag. If the other process is currently running and not stalled out, this function exits because the other process will eventually get to the newly added frame. However, if the other process is stalled out then it must be manually restarted. To do this, the function that primarily implements the other process, `can_tx_send_next_frame`, is called.

```
void can_tx_send_next_frame(CAN_CHANNEL chan);
```

This function pops data off the transmit ring. If no data is available then it sets the stall flag and does not transmit any more data, awaiting restarting by `can_transmit_frame`. Otherwise, the stall flag is cleared and the available frame is written to the transmit registers of the appropriate CAN controller, and then this function exits. This function is then called again by the transmit ISR when the CAN controller is ready to transmit another frame.

Distribution Layer

Concept

The Distribution Layer assembles data from remote locations on a processor into complete CAN frames, and distributes data from complete frames to such locations. Multiple data type combinations, known as layouts, are supported, and data quantities are accessed through getter and setter functions. Frame descriptor structures record a particular layout and which getter and setter functions are used to populate or disseminate the corresponding data. Data transmission with this system can be easily scheduled with the system's main task scheduler. Lists of frame descriptors are used for distributing received frames and handing Remote Transmit Request (RTR) frames.

Implementation

Functions

Rather than reading and writing the memory locations on quantities directly, quantities are accessed through getter and setter functions. The reason for this is twofold. First, there are some situations where it might not be safe to simply read or write a quantity due to a race condition or other problem. In these situations, a wrapper function is can be made to handle these conditions and safely access the quantity in question. Second, many quantities are not transmitted over the wire in the same format as they are used internally on the microcontroller, and conversion between formats can be quite expensive. The use of a wrapper allows conversion between formats only when necessary.

Specifically, getter and setter functions are accessed frequently via what are known as function pointers. A getter function pointer is the address of a function that takes no arguments and returns data of the given type, and a setter function pointer is the address of a function that has a void return type and takes a single argument of the given type.

Layouts and Frame Descriptors

CAN frames support payload sizes of up to eight bytes. That space could be divided and used in many different ways. For example, it could be used for a single 64-bit double-word floating point number, two single-word integers, or a single-word integer and two short integers. These different, possibly mixed data type, payload configurations are called payload layouts, and a number of different layouts are supported with the ability to easily add more.

Clearly, in order to get data where it needs to go, we most associate getter and setter functions with these layouts. To do this we use frame descriptors in the FRAME_DESC structure.

```
typedef struct can_frame_descriptor{
  int           addr : 11;
  CAN_CHANNEL   chan;
  char          rtr  :  1;
  CAN_LAYOUT    frame_layout;
  CAN_VV_PTR    ptr1;
  CAN_VV_PTR    ptr2;
  CAN_VV_PTR    ptr3;
  CAN_VV_PTR    ptr4;
  CAN_VV_PTR    ptr5;
  CAN_VV_PTR    ptr6;
  CAN_VV_PTR    ptr7;
  CAN_VV_PTR    ptr8;
} CAN FRAME DESC;

typedef void(*CAN VV PTR)(void);
```

```
typedef enum can_layout{
  CAN_LAYOUT_D,
  CAN_LAYOUT_FF,
  CAN_LAYOUT_II,
  CAN_LAYOUT_FI,
  CAN_LAYOUT_ISS
}CAN_LAYOUT;
```

This structure is similar to the CAN_FRAME structure in that it too has addr, chan, and rtr fields of the same meaning. However, new are the frame_layout and ptr1 through ptr8 fields. Currently there are five available layouts as shown above in the CAN_LAYOUT structure, where the suffixes indicate layout contents. D indicates a double word floating point value (8 bytes), F indicates a single word floating point value (4 bytes), I indicates a single word integer value (4 bytes), and S indicates a short word integer value (2 bytes).

The ptr# fields are used to store the addresses of the getter and setter functions for use with the given layout. There are eight available fields because the smallest supported type is one byte, leading to eight quantities. These fields are of type void-void function pointer because no single type can match all of the different getter and setter functions used, so void-void was chosen as it is the most generic possible function pointer type.

The use of void-void types clearly makes compile time type checking impossible if the user is to simply assign the addresses of their getter and setter functions directly to a frame descriptor. To solve this problem, the user never directly accesses the function pointer fields of the frame descriptor. Instead, for each layout a set of functions will be created to populate frame descriptors for incoming and outgoing frames. This way, the prototype of these population functions can be used to enforce compile-time type checking and keep the user safe. Example function prototypes for the layout CAN_LAYOUT_FI are below.

```
void can_set_tx_descriptor_fi(CAN_FRAME_DESC* frame_desc,int addr,CAN_CHANNEL
chan,CAN_TX_GETTER_FLOAT g_f1,CAN_TX_GETTER_INT g_i1
  );
void can_set_rx_descriptor_fi(CAN_FRAME_DESC* frame_desc,int addr,
  CAN_RX_SETTER_FLOAT s_f1,CAN_RX_SETTER_INT s_i1
  );
```

Initialization

First, assure that the Transfer Layer has been initialized correctly. If the distribution layer is to be used for *automatically* distributing frames upon receipt, ensure the Transfer Layer is set to automatic dispatch mode. Next, if you wish to use receive and distribution functionality, it is necessary to create a null terminated list of frame descriptors which you wish to receive. Similarly, if you wish to use Remote Transmit Request (RTR) functionality, you must create another null terminated list of the frame descriptors which you wish to be available for RTR transmission. Ensure that all of these descriptors are properly initialized with descriptor population functions as mentioned in the prior section. Now, if you are using either of these, inform the CAN module of these lists by calling `can_rx_set_descriptors`. The null pointer, 0, can be used in place of the list address for a feature which is not desired.

```
void can_rx_set_descriptors(CAN_FRAME_DESC ** rx_descriptors,CAN_FRAME_DESC
** rtr_descriptors);
```

This function stores the addresses of these two given lists in can_rx.c for later use. Note that the receive descriptor list contains all of the information necessary to configure the CAN controller's acceptance filter. The acceptance filter is a hardware filter which allows the user to select which CAN IDs make it through for user processing. This can allow massive processor cycle savings by avoiding processing frames for which you are not the intended recipient. However, acceptance filter configuration has been a low priority and has not been implemented yet. However, this is where it would get called from. Note that this only results in higher processor usage in the event of unintended messages, but not incorrect behavior in the normal case.

Earlier, it was pointed out that there was another field in the receive channel configuration structure, called `descriptors`, which should now make sense. However, that would be misleading. The field in that structure is in fact vestigial and should have been deleted. The receive descriptor list was moved from being channel local to being global among all channels for two reasons. First, it was desirable to separate the configuration of the two layers and making the list global accomplished that goal. However, more importantly, adding RTR support was a late-added feature. However, as RTR frames are ultimately transmit oriented, not receive oriented, which channel they transmit out over is defined in a field of the frame descriptor itself, and so therefore it would not make sense for an RTR list to be associated with any particular channel. Therefore, it was seen as a cleaner solution to keep both lists as similar as possible, and that was accomplished by making both lists global.

Now, although it is not needed to assemble them into a list, allocate all frame descriptors you wish to transmit with this layer, and populate them with the aforementioned functions as appropriate.

Assembly and Transmission

Transmitting a frame descriptor is done through either the `can_transmit` or `can_transmit_alt` functions.

```
int   can_transmit(CAN_FRAME_DESC * fd);
int   can_transmit_alt(CAN_FRAME_DESC * fd,CAN_CHANNEL chan, char rtr);
```

`can_transmit` is the original function for transmitting frame descriptors, while `can_transmit_alt` was added in order to support RTR transmissions and override the `chan` and `rtr` fields of a frame descriptor. In practice, `can_transmit` simply copies the `chan` and `rtr` fields from its given frame descriptor and then calls `can_transmit_alt` with those values to do the real work. This function processes the given frame descriptor based on its `layout` field, interprets all pointers as getter functions because this is a transmit operation that is collecting data, calls each function, and stores each return value in the correct area of a frame payload. If the frame is RTR then no data is collected. The frame's `chan` and `rtr` fields are then filled in according to the function arguments, and the assembled frame is passed the Transfer Layer's `can_transmit_frame` function for transmittal over the bus.

In general, tasks do not necessarily control when their data is sent out over the CAN bus. The reason for this is that CAN bandwidth is scarce and must be conserved, so it may not be desirable for tasks to transmit values on every execution cycle. In order to efficiently utilize bandwidth, it is necessary to coordinate the execution of multiple tasks and transmittal of multiple frames. To do this, CAN transmission scheduling is integrated into the processor's main task execution scheduler. As the task scheduler is based around executing void-void function pointers, CAN transmission must adhere to this scheme as well. Therefore, a frame that is scheduled by the task scheduler must have an associated wrapper function that takes no arguments and returns void. The wrapper can then be inserted into the task scheduler. The primary function of the wrapper's code body is to call the function `can_transmit` with the desired frame descriptor.

Receipt and Distribution

The frame distribution process begins when the Transfer Layer passes a newly received frame to the function `can_rx_dispatch_frame`.

```
void can_rx_dispatch_frame(CAN_FRAME * frame);
```

First, the function searches the RTR or receive descriptor list for a matching CAN ID, depending on whether the frame is or is not an RTR frame, respectively. If no match is found then the process simply ends and the frame is discarded. If an RTR match is found, the matched frame descriptor is passed to can_transmit such that the RTR request is responded to. Otherwise, when a receive descriptor is matched, the data in the received frame is distributed according to the layout and setter functions in the matched frame descriptor.

Limitations and Future Work

Function performance overhead

Global receive descriptor list

Mixed layers Evolution – split to two modules

Single transmit buffer and high priority transmissions

Acceptance filter

Usage Scenarios

The layers of the CAN module are designed such that while they can be used in whole to fulfill their basic usage scenarios, they are also capable of being used in part and configured in alternate manners in order serve advanced functions. Some possible usage scenarios which have been used in the past are described below.

Bidirectional Transfer

The basic normal use of the transfer layer is to provide bidirectional data transfer with a separate buffer for each direction of each channel in use.

Shared Receive Buffers

It is also possible to configure all channels to use the same receive ring buffer. This is useful in two situations. First, in some situations it is necessary or desirable to know the temporal order frames arrived in without recording timestamps. By putting them into a single buffer they are stored in order of arrival. Second, some applications, such as a CAN router, consume all CAN frames in a single location. Storing all frames in a single buffer provides an automatic means of consolidating frames so that they can all be accessed in the same location. This is also useful in the example of the CAN bus probe which allows the user to monitor communications over CAN busses.

Automatic Distribution

The normal usage scenario for the distribution layer is to have data distribution to occur immediately upon receipt.

Delayed Distribution

However, sometimes the data distribution mechanism is desired, but one wants to be able to control the timing of distribution. This can be done by using the Transfer Layer in manual dispatch mode such that received frames go into a ring. Then, simply pop frames off that ring and pass them to `can_rx_dispatch_now` when you want distribution to occur. This behavior was useful, for

example, when an ARM7 was used as the main brain and we wanted to avoid race conditions without mirroring all of our data to allow for asynchronous modifications in response to events on the CAN bus.