```c
/*+
  <INTRO>

  MODULE clkSeq - Clock Sequencer Module Interface Package

  DESCRIPTION:
    Contains the implementation of all the low level VxWorks functions
    which are used to directly access the Clock Sequencer Hardware
    through VME.

    External routines:

    Internal routines:

  </INTRO>

  AUTHOR:

    Laura Paterno
    D0 Project Department
    Particle Physics Division
    Fermilab

  CREATION DATE: 01-Sept-1998

  MODIFICATIONS:

*/

/* VxWorks headers */

#include <vxWorks.h>
#include <sysLib.h>
#include <vme.h>
#include <vxLib.h>
#include <logLib.h>
#include <taskLib.h>

/* Standard headers */

#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h> /* Temporary */
#include <string.h>

/* Local headers */

#include <clkSeq.h>

#define CKN_BUCKET_LO 1                 /* Low End Of RF Bucket Range */
#define CKN_BUCKET_HI 1113              /* High End Of RF Bucket Range */
#define CKN_NUM_STATIC 12               /* Number of Static Timelines */
#define CKN_NUM_DYNAMIC 11              /* Number of Dynamic Timelines */
#define CKN_NUM_REFERENCE 3             /* Number of Reference Timelines */
#define CKN_BUFFER_SIZE 4096            /* Buffer size in words */
#define CKN_NUM_TIMELINES CKN_NUM_STATIC + CKN_NUM_DYNAMIC + CKN_NUM_REFERENCE

/*
   Converts the ID used to in calling the various routines into a pointer to
   the actual structure need by the routines.
*/
#define CONVERT_ID_TO_PTR(seq, so) so = (Ck_Seq_Object_t*) seq

/* Shifts into the appropriate position the value of the Next Address field */
#define SET_NEXT_ADDRESS(x, val) (x) = ((x & 0x000F) | (val << 4))

/* Clock Sequencer Static and Dynamic Buffer */
typedef UINT16 Ck_Buffer_t[CKN_BUFFER_SIZE];

/*
   The bit positions of the Control/Status register for both reading and
   writing to the register.
```



```c
*/
typedef enum {
  R_ERROR_SUM = 0,
  R_CLOCK_ON = 1,
  R_OUTPUT_ENABLE = 2,
  R_STEP_MODE = 3,
  R_FREE_RUN_SYNC = 4,
  R_PROG_TRIGGER_L = 5,
  R_CLEAR_ERRORS = 6,
  R_RESET_SEQ = 7,
  R_GROUP_A_ON = 8,
  R_GROUP_B_ON = 9,
  R_SEL_GR_A_B = 10
} Ck_Read_Seq_t;

/* The bit positions for the Error register for reading the register */
typedef enum {
  CKE_SEQ_HALTED = 0,
  CKE_SYNC_MISSING = 1,
  CKE_BC_PHASE = 2,
  CKE_SEQUENCER_HOLD = 3,
  CKE_CLOCK_PARITY = 4,
  CKE_P_CLOCK_MISSING = 5,
  CKE_M_CLOCK_MISSING = 6
} Ck_Error_Bits_t;

/*
   The bit positions in the Next Address buffer for setting the SYNC_REF,
   BC_REF, and BC_TRIG timeline values.
*/
typedef enum {
  CKE_SYNC_REF = 12,
  CKE_BC_REF = 13,
  CKE_PARITY = 14,
  CKE_BC_TRIG = 15
} Ck_Ref_Lines_t;

typedef enum {
  CKE_DYNAMIC_PARITY = 11
} Ck_Dyn_Lines_t;

/*
   Will hold a copy of all the timelines so as not to interfere with
   the operations of the clock . There is one extra line for holding the parity
   for the static timelines.
*/
typedef Timeline Ck_Timeline_List_t[CKN_NUM_TIMELINES+1];

/* VME Clock Sequencer definition */
typedef struct {
  UINT16 _state;                          /* Control/Status Register */
  UINT16 filler1[CKN_BUFFER_SIZE-1];      /* Filler between C/S and Errors */
  UINT16 _errors;                         /* Error Register */
  UINT16 filler2[(CKN_BUFFER_SIZE*3)-1]; /* Filler between Errors & N. Addr */
  Ck_Buffer_t _nAddr;                     /* Next Address Buffer */
  Ck_Buffer_t _sBuff;                     /* Static Buffer */
  Ck_Buffer_t _dBuffA;                    /* Dynamic Buffer A */
  Ck_Buffer_t _dBuffB;                    /* Dynamic Buffer B */
} Ck_Sequencer_t;

/* Local Copy Sequencer definition */
typedef struct {
  UINT16 _state;
  UINT16 _errors;
  Ck_Timeline_List_t* _tLines;
} Ck_Local_Seq_t;

/* Structure used to store information which will be used in all routines */
typedef struct {
  long _vmeAddr;
  Ck_Sequencer_t* _sVME;
  Ck_Local_Seq_t* _sLocal;
  Ck_Buffer_t* buff;
```

```
} Ck_Seq_Object_t;

void ckZeroBuffer(Ck_Buffer_t* b) {
  int i;

  /* Zero out the timeline buffers */
  for (i=0; i < CKN_BUFFER_SIZE; i++)
    (*b)[i] = 0;
}
```



```
/*
  FUNCTION: clkSeqStateGet

  PURPOSE: Copies the VME values of the Control/Status register into the local
  copy.

  INPUTS:
    seq - pointer the structure used to hold the Control/Status register.

  RETURNS:
    CKS_SUCCESS - success
 */
int clkSeqStateGet(Ck_Seq_Object_t* seq) {

  if (memcpy(&seq->_sLocal->_state, &seq->_sVME->_state, sizeof(UINT16))
      == NULL)
    return CKS_MEMCPY_ERROR;
  CLEAR_BIT_VALUE(seq->_sLocal->_state, R_CLEAR_ERRORS);
  CLEAR_BIT_VALUE(seq->_sLocal->_state, R_RESET_SEQ);

  return (CKS_SUCCESS);
}
```

```
/*
  FUNCTION: clkSeqStateSet

  PURPOSE: Copies the local values of the Control/Status register into the VME
  memory.

  INPUTS:
    seq - pointer the structure used to hold the Control/Status register.

  RETURNS:
    CKS_SUCCESS - success
 */
int clkSeqStateSet(Ck_Seq_Object_t* seq) {

  if (memcpy(&seq->_sVME->_state, &seq->_sLocal->_state, sizeof(UINT16))
        == NULL)
    return CKS_MEMCPY_ERROR;

  CLEAR_BIT_VALUE(seq->_sLocal->_state, R_PROG_TRIGGER_L);
  CLEAR_BIT_VALUE(seq->_sLocal->_state, R_ERROR_SUM);
  CLEAR_BIT_VALUE(seq->_sLocal->_state, R_CLEAR_ERRORS);
  CLEAR_BIT_VALUE(seq->_sLocal->_state, R_RESET_SEQ);

  return (CKS_SUCCESS);
}
```



```
/*
  FUNCTION: clkSeqNextAddrInit

  PURPOSE: Initializes the VME Next Address buffer memory so that timelines
  can be correctly read out from the Clock Sequencer.

  INPUTS:
    seq - pointer the structure used to hold the Control/Status register.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while copying the memory from the
    temporary copy to the VME memory of the Next Address buffer.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqNextAddrInit(Ck_Seq_Object_t* theSeq) {
  int bucket;

  if (theSeq != NULL) {
    ckZeroBuffer(theSeq->buff);

    /* Create temporary Next Address Block and then copy it into VME memory */
    for (bucket = 0; bucket < CKN_BUCKET_HI-1; bucket++) {
      (*theSeq->buff)[bucket] = bucket+1;
    }

    /* Set the last bucket position to return to the first position */
    (*theSeq->buff)[CKN_BUCKET_HI-1] = 0;
    for (bucket = CKN_BUCKET_HI; bucket < CKN_BUFFER_SIZE; bucket++)
      (*theSeq->buff)[bucket] = 0;

    /* Copy the temporary copy back into the VME memory */
    for (bucket=0; bucket<CKN_BUFFER_SIZE; bucket++) {
      theSeq->_sVME->_nAddr[bucket] = (*theSeq->buff)[bucket];
      taskDelay(sysClkRateGet()/80);
    }
    return CKS_SUCCESS;
 }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqStateBitSet

  PURPOSE: Sets the appropriate bit in the local copy of the Control/Status
  register so that it can be copied to VME when requested.

  INPUTS:
    seq - ID of the structure used to hold the Control/Status register.

  RETURNS:
    CKS_SUCCESS - success
    CKS_ALREADY_IN_STATE - not an error. Success but nothing is done because
    bit is already set.
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqStateBitSet(ClkSeqId_t seq, int theBit) {
  Ck_Seq_Object_t* theSeq;
  int seqStat = CKS_SUCCESS;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {
    if (theBit != R_ERROR_SUM)
      CLEAR_BIT_VALUE(theSeq->_sLocal->_state, R_ERROR_SUM);

    if (GET_BIT_VALUE(theSeq->_sLocal->_state, theBit) == 0)
      SET_BIT_VALUE(theSeq->_sLocal->_state, theBit);
    else
      return CKS_ALREADY_IN_STATE;

    /* Set the value in VME */
    if ((seqStat = clkSeqStateSet(theSeq)) < CKS_SUCCESS)
      return seqStat;

    /* Read values back in to get the current state */
    return clkSeqStateGet(theSeq);
  }
  else
    return CKS_INVALID_POINTER;
}
```



```
/*
  FUNCTION: clkSeqStateBitClear

  PURPOSE: Clears the appropriate bit in the local copy of the Control/Status
  register so that it can be copied to VME when requested.

  INPUTS:
    seq - ID of the structure used to hold the Control/Status register.

  RETURNS:
    CKS_SUCCESS - success
    CKS_ALREADY_IN_STATE - not an error. Success but nothing is done because
    bit is already clear.
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqStateBitClear(ClkSeqId_t seq, int theBit) {
  Ck_Seq_Object_t* theSeq;
  int seqStat;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {
    if (theBit != R_ERROR_SUM)
      CLEAR_BIT_VALUE(theSeq->_sLocal->_state, R_ERROR_SUM);

    if (GET_BIT_VALUE(theSeq->_sLocal->_state, theBit) != 0)
      CLEAR_BIT_VALUE(theSeq->_sLocal->_state, theBit);
    else
      return CKS_ALREADY_IN_STATE;

    /* Clear the value in VME */
    if ((seqStat = clkSeqStateSet(theSeq)) < CKS_SUCCESS)
      return seqStat;

    /* Read values back in to get the current state */
    return  clkSeqStateGet(theSeq);
  }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqReferenceLinesLoad

  PURPOSE: Loads the SYNC_REF, BC_REF and BC_TRIG timelines into the Next
  Address Buffer. It also loads in the parity for the Static Timelines.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqReferenceLinesLoad(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* theSeq;
  int i,j, seqOn, seqStat;
  int tLine;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {
    /* Turn the Clock off if it is on */
    if ((seqOn = GET_BIT_VALUE(theSeq->_sLocal->_state, R_CLOCK_ON)) != 0) {
      seqStat = clkSeqClockOff(seq);
      if (seqStat < CKS_SUCCESS) return seqStat;
    }

    /* Zero out the copy of the buffer */
    ckZeroBuffer(theSeq->buff);

    /* Copy the Current Next Address Buffer into temporary copy */
    for (i=0; i<CKN_BUFFER_SIZE; i++) {
      (*theSeq->buff)[i] = theSeq->_sVME->_nAddr[i];
      taskDelay(sysClkRateGet()/80);
    }

    /* Set the new SYNC_REF, BC_REF and BC_TRIG values */
    for (i = SYNC_REF; i <= BC_TRIG+1; i++) {
      /* Sets tLine to the correct bit position to modify in Next Address */
      if (i == SYNC_REF) tLine = CKE_SYNC_REF;
      else if (i == BC_REF) tLine = CKE_BC_REF;
      else if (i == BC_TRIG) tLine = CKE_BC_TRIG;
      else tLine = CKE_PARITY;
      for (j = CKN_BUCKET_LO; j < CKN_BUCKET_HI; j++) {
        if (timelineGet((*theSeq->_sLocal->_tLines)[i], j) != 0) {
          SET_BIT_VALUE((*theSeq->buff)[j-1], tLine);
        }
        else {
          CLEAR_BIT_VALUE((*theSeq->buff)[j-1], tLine);
        }
      }
    }

    /*
    for (i=CKN_BUCKET_LO; i <10; i++)
        printf("Buffer %d: %d\n", i-1, (*theSeq->buff)[i-1]);
        */

    /* Copy the temporary copy back into the VME memory */
    for (i=0; i<CKN_BUFFER_SIZE; i++) {
      theSeq->_sVME->_nAddr[i] = (*theSeq->buff)[i];
      taskDelay(sysClkRateGet()/80);
    }

    /* If the clock was off turn it back on */
    if (seqOn == 1)
      return clkSeqClockOn(seq);
    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
```



```
}
```

```
/*
  FUNCTION: clkSeqDynamicLinesLoad

  PURPOSE: Loads the Dynamic Timelines with the new timelines

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqDynamicLinesLoad(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* theSeq;
  int i,j, buffer, seqStat, parity;
  int tLine;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {

    /* Zero out the copy of the buffer */
    ckZeroBuffer(theSeq->buff);

    /* Convert the Local Copy of the Dynamic Lines into a Buffer */
    for (j = CKN_BUCKET_LO; j<= CKN_BUCKET_HI; j++) {
      parity = 0;
      for(i=T12; i<=T22; i++) {
        tLine = i - (CKN_NUM_DYNAMIC + 1);
        if (timelineGet((*theSeq->_sLocal->_tLines)[i], j) != 0) {
          SET_BIT_VALUE((*theSeq->buff)[(j-1)], tLine);
          parity = parity + 1;
        }
        else {
          CLEAR_BIT_VALUE((*theSeq->buff)[(j-1)], tLine);
        }
      }

      if (parity > 0) {
        if ((parity % 2) != 0)
          SET_BIT_VALUE((*theSeq->buff)[(j-1)], CKE_DYNAMIC_PARITY);
        else
          CLEAR_BIT_VALUE((*theSeq->buff)[(j-1)], CKE_DYNAMIC_PARITY);
      }
      else
          CLEAR_BIT_VALUE((*theSeq->buff)[(j-1)], CKE_DYNAMIC_PARITY);
    }
    /*
    for (i=CKN_BUCKET_LO; i <10; i++)
      printf("Buffer %d: %d\n", i-1, (*theSeq->buff)[i-1]);
      */

    /* Figure out which buffer is off */
    if ((buffer = GET_BIT_VALUE(theSeq->_sLocal->_state, R_GROUP_A_ON)) != 0) {
      /* Copy the temporary buffer into the buffer which is off */
      for (i=0; i< CKN_BUFFER_SIZE; i++) {
        theSeq->_sVME->_dBuffB[i] = (*theSeq->buff)[i];
      taskDelay(sysClkRateGet()/80);

      }

    }
    else {
      for (i=0; i< CKN_BUFFER_SIZE; i++) {
        theSeq->_sVME->_dBuffA[i] = (*theSeq->buff)[i];
      taskDelay(sysClkRateGet()/80);
      }
    }

    /*
    if (buffer != 0) {
```



```
      for (i=CKN_BUCKET_LO; i <10; i++)
        printf("Buffer B %d: %d\n", i-1, theSeq->_sVME->_dBuffB[i-1]);
    }
    else {
      for (i=CKN_BUCKET_LO; i <10; i++)
        printf("Buffer A %d: %d\n", i-1, theSeq->_sVME->_dBuffA[i-1]);
    }
    */

    /* Turn off the appropriate buffer and select the new buffer */
    if (buffer != 0) {
      if ((seqStat = clkSeqStateBitClear(theSeq, R_GROUP_A_ON)) < CKS_SUCCESS)
        return seqStat;
      if ((seqStat = clkSeqStateBitSet(theSeq, R_GROUP_B_ON)) < CKS_SUCCESS)
        return seqStat;
      return clkSeqStateBitClear(theSeq, R_SEL_GR_A_B);
    }
    else {
      if ((seqStat = clkSeqStateBitClear(theSeq, R_GROUP_B_ON)) < CKS_SUCCESS)
        return seqStat;
      if ((seqStat = clkSeqStateBitSet(theSeq, R_GROUP_A_ON)) < CKS_SUCCESS)
        return seqStat;
      return clkSeqStateBitSet(theSeq, R_SEL_GR_A_B);
    }
  }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqStaticLinesLoad

  PURPOSE: Loads the static timelines with new timeline values.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqStaticLinesLoad(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* theSeq;
  int i,j, seqOn, seqStat, parity;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {
    /* If the clock is on turn it off */
    if ((seqOn = GET_BIT_VALUE(theSeq->_sLocal->_state, R_CLOCK_ON)) != 0) {
      seqStat = clkSeqClockOff(seq);
      if (seqStat < CKS_SUCCESS) return seqStat;
    }

    ckZeroBuffer(theSeq->buff);

    /* Convert the Local Copy of the Static Lines into a Buffer */
    for (j = CKN_BUCKET_LO; j <= CKN_BUCKET_HI; j++) {
      parity = 0;
      for (i = T0; i<= T11; i++) {
        if (timelineGet((*theSeq->_sLocal->_tLines)[i], j) != 0) {
          SET_BIT_VALUE((*theSeq->buff)[j-1], i);
          parity = parity + 1;
        }
        else
          CLEAR_BIT_VALUE((*theSeq->buff)[j-1], i);
      }
      if (parity > 0) {
        if ((parity % 2) != 0)
          timelineSet((*theSeq->_sLocal->_tLines)[BC_TRIG+1], j, j);
        else
          timelineClear((*theSeq->_sLocal->_tLines)[BC_TRIG+1], j, j);
      }
      else
        timelineClear((*theSeq->_sLocal->_tLines)[BC_TRIG+1], j, j);
    }
    /*
    for (i=CKN_BUCKET_LO; i <10; i++)
      printf("Buffer %d: %d\n", i-1, (*theSeq->buff)[i-1]);
      */

    /* Copy the temporary copy of the buffer into the VME buffer */
    for (i=0; i< CKN_BUFFER_SIZE; i++) {
      theSeq->_sVME->_sBuff[i] = (*theSeq->buff)[i];
      taskDelay(sysClkRateGet()/80);

    }
    if ((seqStat = clkSeqReferenceLinesLoad(seq)) < CKS_SUCCESS)
      return seqStat;

    /* If the clock was off turn it back on */
    if (seqOn == 1)
      return clkSeqClockOn(seq);
    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
}
```



```
/*
  FUNCTION: clkSeqInit

  PURPOSE: Initializes the structures used to control the Clock Sequencer. It
  also sets the Sequencer to a default state by resetting it, setting the Next
  Address buffer appropriately and zeroing all the timeline buffers.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_NO_MEMORY - unable to allocate memory for a structure.
    CKS_BUS_TO_LOCAL_ADDRESS - unable to convert the VME base address to a
    local address.
    CKS_INVALID_VME_ADDRESS - the VME address specified is not accessible.
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqInit(ClkSeqId_t* seq, const long vmeAddress)
{
  Ck_Seq_Object_t* theSeq;
  int seqStat, i;
  int localAddress;
  char tmp;

  /* Allocate the internal Sequencer Object */
  theSeq = (Ck_Seq_Object_t*) malloc(sizeof(Ck_Seq_Object_t));
  if (theSeq == NULL) {
    return (CKS_NO_MEMORY);
  }

  /* Map the VME base address and probe both register blocks */
  if (sysBusToLocalAdrs(VME_AM_STD_SUP_DATA, (char *) vmeAddress,
                        (char**) &localAddress) == ERROR) {
     return (CKS_BUS_TO_LOCAL_ADDRESS);
  }

  /* Assign the Local Address to the appropriate structure */
  theSeq->_sVME = (Ck_Sequencer_t *) localAddress;

 /* Check that the memory location is actually addressable */
  if (vxMemProbe((char*) theSeq->_sVME, VX_READ, 1, &tmp) == ERROR) {
    return (CKS_INVALID_VME_ADDRESS);
    }

  *seq = (ClkSeqId_t) theSeq;

  /* Allocate the local copy structure for the Sequencer */
  theSeq->_sLocal = (Ck_Local_Seq_t *) malloc(sizeof(Ck_Local_Seq_t));
  if (theSeq->_sLocal == NULL) {
    theSeq->_vmeAddr = 0;
    free(theSeq);
    return (CKS_NO_MEMORY);
  }

  /* Create the timelines. TimelineCreate will zero out the timelines */
  theSeq->_sLocal->_tLines = (Ck_Timeline_List_t *)
    malloc(sizeof(Ck_Timeline_List_t));
  if (theSeq->_sLocal->_tLines == NULL) {
    theSeq->_vmeAddr = 0;
    free(theSeq);
    return (CKS_NO_MEMORY);
  }

  for (i=0; i < CKN_NUM_TIMELINES+1; i++) {
    (*theSeq->_sLocal->_tLines)[i] = timelineCreate();
 }

  /* Initialize Local and VME copies of Sequencer to default state */
  if ((seqStat = clkSeqReset(*seq)) < CKS_SUCCESS)
    return seqStat;
```

```
  if ((seqStat = clkSeqStateGet(theSeq)) < CKS_SUCCESS)
    return seqStat;

  theSeq->buff = (Ck_Buffer_t *) malloc(sizeof(Ck_Buffer_t));
  if (theSeq->buff == NULL) {
    theSeq->_vmeAddr = 0;
    free(theSeq);
    return (CKS_NO_MEMORY);
  }

  ckZeroBuffer(theSeq->buff);

  if ((seqStat = clkSeqClockOff(*seq)) < CKS_SUCCESS)
    return seqStat;

  /* Assign the Next Address Block and Zero All Timelines */
  if ((seqStat = clkSeqNextAddrInit(theSeq)) < CKS_SUCCESS)
    return seqStat;

  ckZeroBuffer(theSeq->buff);

  /* Turn on Dynamic Buffer A and select it and turn off Dynamic Buffer B */
  if ((seqStat = clkSeqStateBitSet(*seq, R_GROUP_A_ON)) < CKS_SUCCESS)
    return seqStat;
  if ((seqStat = clkSeqStateBitSet(*seq, R_SEL_GR_A_B)) < CKS_SUCCESS)
    return seqStat;
  if ((seqStat = clkSeqStateBitClear(*seq, R_GROUP_B_ON)) < CKS_SUCCESS)
    return seqStat;

  /* Zero out all the timeline buffers */
  if ((seqStat = clkSeqStaticLinesLoad(*seq)) < CKS_SUCCESS)
    return seqStat;

  /* Load Dynamic Lines twice to zero both buffers */
  if ((seqStat = clkSeqDynamicLinesLoad(*seq)) < CKS_SUCCESS)
    return seqStat;
  if ((seqStat = clkSeqDynamicLinesLoad(*seq)) < CKS_SUCCESS)
    return seqStat;

  /* Clear all errors resulting from the Reset of the Sequencer */
  return clkSeqErrorsClear(*seq);
}
```



```
/*
  FUNCTION: clkSeqDestroy

  PURPOSE: Destroys the structures used to control the Clock Sequencer.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqDestroy(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* theSeq;
  int tLine;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {

    for (tLine = 0; tLine < CKN_NUM_TIMELINES+1; tLine++) {
      timelineDestroy((*theSeq->_sLocal->_tLines)[tLine]);
    }
    free(theSeq->buff);
    free(theSeq->_sLocal->_tLines);
    free(theSeq->_sLocal);
    free(theSeq);
    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqReset

  PURPOSE: Resets the Clock Sequencer to its hardware default state.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqReset(ClkSeqId_t seq)
{
  int seqStat;

  if ((seqStat = clkSeqStateBitSet(seq, R_RESET_SEQ)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```



```
/*
  FUNCTION: clkSeqTimelineShow

  PURPOSE: Shows the specified timeline. Strictly a diagnostic tool used on the
  front ends.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqTimelineShow(ClkSeqId_t seq, Cke_Timelines_t tnum)
{
  int i, spos, epos, cval, sval;
  Ck_Seq_Object_t* theSeq;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL) {
    i = 1;
    sval = timelineGet((*theSeq->_sLocal->_tLines)[tnum], i);
    spos = epos = i;
    while (i < CKN_BUCKET_HI) {
      i++;
      while (((cval = timelineGet((*theSeq->_sLocal->_tLines)[tnum], i))
              == sval) && (i < CKN_BUCKET_HI)) {
        epos = i;
        i++;
      }
      if (i == CKN_BUCKET_HI) {
        if ((cval = timelineGet((*theSeq->_sLocal->_tLines)[tnum], i))
            == sval) {
          epos = i;
          if (sval == 0)
            logMsg("Timeline T%d DOWN FROM %d TO %d\n",tnum,spos,epos,0,0,0);
          else
            logMsg("Timeline T%d UP FROM %d to %d\n", tnum,spos, epos,0,0,0);
        }
        else {
          if (sval == 0)
            logMsg("Timeline %d DOWN FROM %d TO %d\n", tnum, spos, epos,0,0,0);
          else
            logMsg("Timeline %d UP FROM %d to %d\n", tnum, spos, epos,0,0,0);
          spos = epos = i;
          sval = cval;
          if (sval == 0)
            logMsg("Timeline %d DOWN FROM %d TO %d\n", tnum, spos, epos,0,0,0);
          else
            logMsg("Timeline %d UP FROM %d to %d\n", tnum, spos, epos,0,0,0);
        }
      }
      else {
        if (sval == 0)
          logMsg("Timeline %d DOWN FROM %d TO %d\n", tnum, spos, epos,0,0,0);
        else
          logMsg("Timeline %d UP FROM %d to %d\n", tnum, spos, epos,0,0,0);
        spos = epos = i;
        sval = cval;
      }
    }

    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqTimelineGet

  PURPOSE: Gets the specified timeline from the local copy of the timelines
  and returns it in t.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqTimelineGet(ClkSeqId_t seq, Cke_Timelines_t tnum, Timeline t)
{
  Ck_Seq_Object_t* so;

  if (seq != NULL) {
    CONVERT_ID_TO_PTR(seq, so);
    timelineCopy((*so->_sLocal->_tLines)[tnum], t);

    return CKS_SUCCESS;
  } /* seq != NULL */

  else
    return CKS_INVALID_POINTER;
}
```



```
/*
  FUNCTION: clkSeqTimelinePut

  PURPOSE: Puts the specified timeline into the local copy of the timelines.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */

int clkSeqTimelinePut(ClkSeqId_t seq,
                      Cke_Timelines_t tnum,
                      Timeline t)

  {
  Ck_Seq_Object_t* theSeq;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL)
    {
    timelineCopy(t, (*theSeq->_sLocal->_tLines)[tnum]);
/*     if (tnum <= T11) */
/*       { return clkSeqStaticLinesLoad(seq); } */
/*     else if (tnum <= T22) */
/*       { return clkSeqDynamicLinesLoad(seq); } */
/*     else */
/*       { return clkSeqReferenceLinesLoad(seq); } */
/*     return clkSeqErrorsClear(seq); */
    return (CKS_SUCCESS);
    }
  else
    return CKS_INVALID_POINTER;
  }

int clkSeqTimelinesLoad(ClkSeqId_t seq)

  {
  int seqStat;
  Ck_Seq_Object_t* theSeq;

  CONVERT_ID_TO_PTR(seq, theSeq);
  if (theSeq != NULL)
    {
    if ((seqStat = clkSeqStaticLinesLoad(seq)) < CKS_SUCCESS)
      return seqStat;
    if ((seqStat = clkSeqDynamicLinesLoad(seq)) < CKS_SUCCESS)
      return seqStat;
    if ((seqStat = clkSeqReferenceLinesLoad(seq)) < CKS_SUCCESS)
      return seqStat;
    return clkSeqErrorsClear(seq);
    }
  else
    return CKS_INVALID_POINTER;
  }
```

```
/*
  FUNCTION: clkSeqClockOn

  PURPOSE: Turns the clock on.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqClockOn(ClkSeqId_t seq)
{
  int seqStat;

  if ((seqStat = clkSeqStateBitSet(seq, R_CLOCK_ON)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```



```
/*
  FUNCTION: clkSeqClockOff

  PURPOSE: Turns the clock off.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqClockOff(ClkSeqId_t seq)
{
  int seqStat;
  if ((seqStat = clkSeqStateBitClear(seq, R_CLOCK_ON)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```

```
/*
  FUNCTION: clkSeqOutputEnable

  PURPOSE: Enables the output of the timelines along the VME backplane.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqOutputEnable(ClkSeqId_t seq)
{
  int seqStat;
  if ((seqStat = clkSeqStateBitSet(seq, R_OUTPUT_ENABLE)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```



```
/*
  FUNCTION: clkSeqOutputDisable

  PURPOSE: Disables the output of the timelines along the VME backplane.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqOutputDisable(ClkSeqId_t seq)
{
  int seqStat;
  if ((seqStat = clkSeqStateBitClear(seq, R_OUTPUT_ENABLE)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```

```
/*
  FUNCTION: clkSeqModeFreerunSet

  PURPOSE: Puts the Clock Sequencer in Freerun mode. This will enable the
  Sequencer to internal generate signals that it normally receives from outside
  sources.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqModeFreerunSet(ClkSeqId_t seq)
{
  int seqStat;
  if ((seqStat = clkSeqStateBitSet(seq, R_FREE_RUN_SYNC)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```



```
/*
  FUNCTION: clkSeqModeSyncSet

  PURPOSE: Puts the Clock Sequencer in Sync mode. This means that it will be
  synchronizing itself to externally generated signals being sent to it.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqModeSyncSet(ClkSeqId_t seq)
{
  int seqStat;
  if ((seqStat = clkSeqStateBitClear(seq, R_FREE_RUN_SYNC)) < CKS_SUCCESS)
    return seqStat;
  return clkSeqErrorsClear(seq);
}
```

```
/*
  FUNCTION: clkSeqErrors set

  PURPOSE: Turns all the error bits on.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqErrorsSet(ClkSeqId_t seq)
{
  int seqStat;
  return (seqStat = clkSeqStateBitSet(seq, R_ERROR_SUM));
}
```



```
/*
  FUNCTION: clkSeqErrorsGet

  PURPOSE: Returns the current value of the error bits register.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    ce - the error bits register of the Clock Sequencer.

    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqErrorsGet(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* so;
  if (seq != NULL) {
    CONVERT_ID_TO_PTR(seq, so);
    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqErrorsClear

  PURPOSE: Clears all the error bits that currently show as errors on the
  Clock Sequencer front panel but may not necessarily be real errors.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqErrorsClear(ClkSeqId_t seq)
{
  return clkSeqStateBitSet(seq, R_CLEAR_ERRORS);
}
```



```
/*
  FUNCTION: clkSeqErrorsShow

  PURPOSE: Diagnostic tool used to display the current errors of the Clock
  Sequencer. It is only usable on the front end.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqErrorsShow(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* so;

  if (seq != NULL) {
    CONVERT_ID_TO_PTR(seq, so);
    logMsg("CURRENT Errors:\n",0,0,0,0,0,0);
    logMsg("==============\n",0,0,0,0,0,0);

    /* Copy VME Errors into Local Errors */
    so->_sLocal->_errors = so->_sVME->_errors;

    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_SEQ_HALTED) != 0)
      logMsg("SEQUENCER HALTED\n",0,0,0,0,0,0);
    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_SYNC_MISSING) != 0)
      logMsg("SYNC MISSING",0,0,0,0,0,0);
    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_BC_PHASE) != 0)
      logMsg("BC PHASE",0,0,0,0,0,0);
    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_SEQUENCER_HOLD) != 0)
      logMsg("SEQUENCER HOLD",0,0,0,0,0,0);
    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_CLOCK_PARITY) != 0)
      logMsg("CLOCK PARITY",0,0,0,0,0,0);
    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_P_CLOCK_MISSING) != 0)
      logMsg("P CLK MISSING",0,0,0,0,0,0);
    if (GET_BIT_VALUE(so->_sLocal->_errors, CKE_M_CLOCK_MISSING) != 0)
      logMsg("M CLK MISSING",0,0,0,0,0,0);

    logMsg("\n",0,0,0,0,0,0);

    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
}
```

```
/*
  FUNCTION: clkSeqCtrlStatGet

  PURPOSE: Gets the current value of the Control/Status register of the Clock
  Sequencer.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqCtrlStatGet(ClkSeqId_t seq, clkCtrlStat_t cs)
{
  Ck_Seq_Object_t* so;
  if (seq != NULL) {
    CONVERT_ID_TO_PTR(seq, so);
    cs = (unsigned long) so->_sLocal->_state;
    return CKS_SUCCESS;
  }
  else
    return CKS_INVALID_POINTER;
}
```



```
/*
  FUNCTION: clkSeqCtrlStatShow

  PURPOSE: Diagnostic tool used on the front ends only to show the current
  state of the Clock Sequencer.

  INPUTS:
    seq - ID of the structure used to control the sequencer.

  RETURNS:
    CKS_SUCCESS - success
    CKS_MEMCPY_ERROR - an error occurred while doing a memory copy.
    CKS_INVALID_POINTER - the ID passed in to the routine was invalid.
 */
int clkSeqCtrlStatShow(ClkSeqId_t seq)
{
  Ck_Seq_Object_t* so;
  int seqStat;

  CONVERT_ID_TO_PTR(seq, so);
  if ((seqStat = clkSeqStateGet(so)) < CKS_SUCCESS)
    return seqStat;

  logMsg("CURRENT STATE:\n",0,0,0,0,0,0);
  logMsg("==============\n",0,0,0,0,0,0);
  logMsg("Clock %s\n",
         (int)(GET_BIT_VALUE(so->_sLocal->_state, R_CLOCK_ON) ? "ON" :
               "OFF"),0,0,0,0,0);
  logMsg("Output %s\n",
         (int)(GET_BIT_VALUE(so->_sLocal->_state, R_OUTPUT_ENABLE) ?
               "ENABLED" : "DISABLED"),0,0,0,0,0);
  logMsg("Mode %s\n",
         (int)(GET_BIT_VALUE(so->_sLocal->_state, R_FREE_RUN_SYNC) ?
               "FREERUN" : "SYNC"),0,0,0,0,0);
  logMsg("Buffer A %s\n",
         (int)(GET_BIT_VALUE(so->_sLocal->_state, R_GROUP_A_ON) ? "ON"
               : "OFF"),0,0,0,0,0);
  logMsg("Buffer B %s\n",
         (int)(GET_BIT_VALUE(so->_sLocal->_state, R_GROUP_B_ON) ? "ON"
               : "OFF"),0,0,0,0,0);
  logMsg("Buffer %s Selected\n",
         (int)(GET_BIT_VALUE(so->_sLocal->_state, R_SEL_GR_A_B) ? "A" :
               "B"), 0,0,0,0,0);
  logMsg("\n",0,0,0,0,0,0);
  return CKS_SUCCESS;
}
```