TEXAS INSTRUMENTS

TVP5020 NTSC/PAL Video Decoder

Programming for the VMI Host Interface

Application Report

March 1999

Mixed-Signal Product

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

List of Figures

List of Tables

TVP5020 NTSC/PAL Video Decoder Programming for the VMI Host Interface

Michael A. Tadyshak

ABSTRACT

This application report provides a complete working example of a C-language program to initialize the TVP5020 NTSC/PAL video decoder using the VMI Bus interface. This Example Initialization Program executes on the TVP56000EVM evaluation module featuring the TVP5020 video decoder and the TVP6000 video encoder. Topics covered include the TVP56000EVM hardware platform, microcontroller-specific aspects, and a detailed description of the source code modules.

1 Introduction

The TVP5020 NTSC/PAL Video Decoder enables a wide range of applications by providing support for each of the following host interfaces:

- I^{2}C (Inter-Integrated Circuit) Bus © 1995 Philips Semiconductors
- VIP (Video Interface Port) © 1994, 1996, 1997 Video Electronic Standards Association
- VMI (Video Module Interface) © 1997 Cirrus Logic

Software development time can be reduced by utilizing this Example Initialization Program. The user can modify and rebuild the source code for quick verification on the TVP56000EVM. Also, the TVP5020-specific source code modules can be used as a reference for software development for the user’s own hardware platform.

2 Hardware Platform

2.1 Block Diagram

This program was tested on the TVP56000EVM (Evaluation Module for the TVP5020 Video Decoder and the TVP6000 Video Encoder). A block diagram of this EVM is shown in Figure 1. The program is executed by the Philips P80C652 Microcontroller. This device is a derivative of the Intel 80C51 microcontroller and is second-sourced by Philips. The P80C652 includes an on-chip I^{2}C Bus controller. The program is stored in the 64 KB flash memory device. This device is an Atmel 29C512. Since this program is a scaled-down version of the complete EVM microcode program, it does not support flash reprogramming or the serial port. In order to install this program on the EVM, it must be programmed into the flash memory device with a PROM programmer. The EVM also has 32 KB of static RAM available for program use.

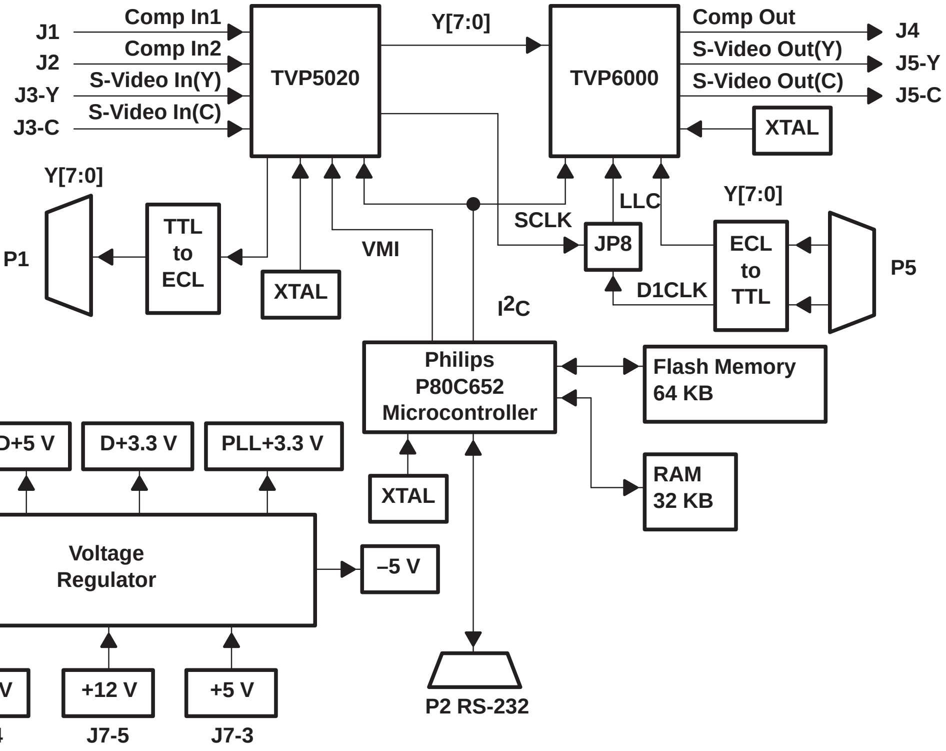


Figure 1. TVP56000EVM Block Diagram

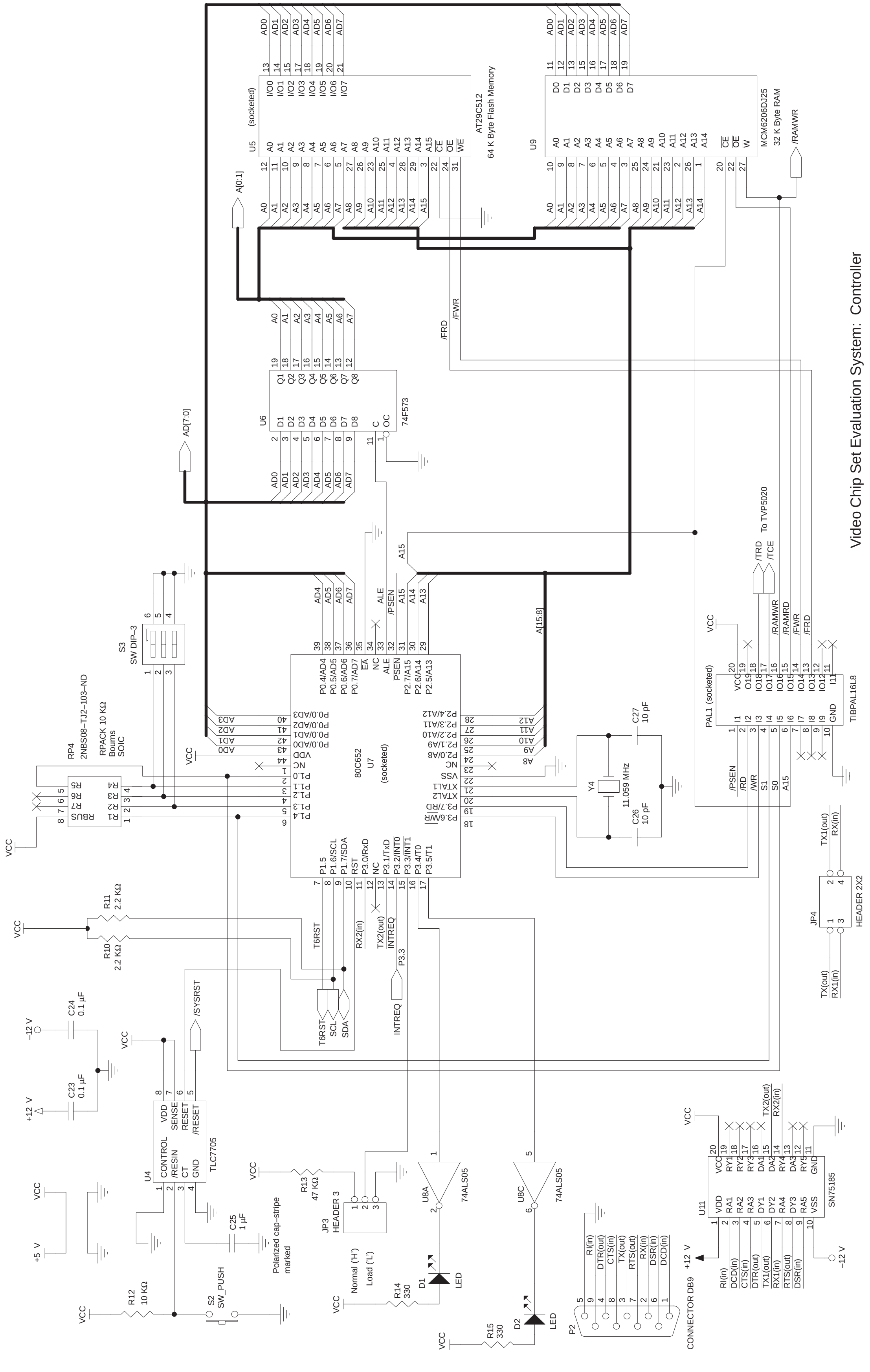


Figure 2. Microcontroller Interface Schematic

2.2 Schematic Diagram – Microcontroller Interface

Figure 2 shows the schematic diagram of the microcontroller and surrounding circuitry. Refer to this diagram to understand some of the hardware-specific aspects of the program. The P80C652 has four 8-bit I/O ports. Since the device is configured to use external memory, the I/O port P0 is used as the multiplexed lower address and data (AD[7–0]), and the I/O port P2 is used to output the upper address bits A[15–8]. I/O ports P1 and P3 have some dedicated functions and are otherwise user-defined, as described in Table 1.

The names in the signal name column are shown exactly as they would appear in the source code. Each is a macro representing a special-function register or a special-function I/O bit as defined for the P80C652. The user-defined I/O bits S0, S1, FLASHJMP, LED1, and LED2 are not used by this program. SW2 and SW3 are used to read the corresponding DIP-switches to select the video mode as shown in Table 2. DIP-switch SW1 is not used. T6RESET provides a software means to reset the TVP6000 video encoder. INTREQ is used to read the interrupt request signal from the TVP5020.

Notice that the I2C clock (SCL) and I2C data (SDA) signals must each be pulled up to V_{CC} with a 2.2 kΩ resistor.

Table 1. Microcontroller I/O Port Utilization

PIN NAME	SIGNAL NAME	DIRECTION	FUNCTION	DEFINITION
P1.0	S0	Output	Flash memory state lsb	User–defined
P1.1	SW1	Input	DIP switch 1	User–defined
P1.2	SW2	Input	DIP switch 2	User–defined
P1.3	SW3	Input	DIP switch 3	User–defined
P1.4	S1	Output	Flash memory state msb	User–defined
P1.5	T6RESET	Output	TVP6000 reset	User–defined
P1.6	SCL	Output	I2C clock	Dedicated I/O
P1.7	SDA	I/O	I2C data	Dedicated I/O
P3.0	RXD	Input	RS–232 receive data	Dedicated I/O
P3.1	TXD	Output	RS–232 transmit data	Dedicated I/O
P3.2	INTREQ	Input	TVP5020 interrupt request	User–defined
P3.3	FLASHJMP	Input	Flash memory jumper	User–defined
P3.4	LED1	Output	LED D1	User–defined
P3.5	LED2	Output	LED D2	User–defined
P3.6	WR	Output	External memory write strobe	Dedicated I/O
P3.7	RD	Output	External memory read strobe	Dedicated I/O

Table 2. DIP-Switch Settings for TVP56000EVM

VIDEO STANDARD	SAMPLING RATE	INDIVIDUAL SWITCHES		
		S3–3	S3–2	S3–1
NTSC	CCIR601	ON	ON	X
NTSC	Square pixel	OFF	ON	X
PAL	CCIR601	ON	OFF	X
PAL	Square pixel	OFF	OFF	X

2.2.1 Memory Mapped VMI Interface

The TVP56000EVM uses a memory-mapped I/O scheme to control the TVP5020's VMI host interface. The 64 kB external data memory space is split in half. The lower 32 kB are the RAM chip (U9). The RAM chip enable is active only when the upper address bit (A15) is 0. See the logic equations for PAL1 in Section 2.3. The S1 and S0 signals are state indicators for the flash memory reprogram process. At all other times S1 = 1 and S0 = 1. The TVP5020 chip enable ($\overline{\text{TCE}}$) is then active when A15 is 1. This corresponds to addresses 8000h – FFFFh, but only addresses 8000h – 8003h are actually used for VMI. The TVP5020 read-enable ($\overline{\text{TRD}}$) goes active when the read strobe ($\overline{\text{RD}}$) from the microcontroller goes active. The TVP5020 write-enable ($\overline{\text{RWR}}$) has the same equation as the RAM write-enable and goes active when the write strobe ($\overline{\text{WR}}$) from the microcontroller goes active. The $\overline{\text{WR}}$ and $\overline{\text{RD}}$ signals are activated when the microcontroller accesses the external data memory ('xdata') space.

The EVM supports VMI timing mode B, for which:

- VC0 is the RDY output and is not connected. Jumper JP12 must be removed.
- VC1 is the $\overline{\text{WR}}$ input and must be jumpered to $\overline{\text{RAMWR}}$ by connecting jumper JP6 across pins 2 and 3.
- VC2 is the $\overline{\text{RD}}$ input and is connected directly to $\overline{\text{TRD}}$ from PAL1.
- VC3 is the $\overline{\text{CE}}$ input and must be jumpered to $\overline{\text{TCE}}$ by connecting jumper JP5 across pins 1 and 2.

On the software side, an array of four bytes named *g_pVMI* is declared in *xdata* space. This *must* be done in a separate source file as shown in the listing for VMI_PORT.C in Section 2.4. In the make file shown in Section 2.5, all variables declared in the file VMI_PORT.C are specified to be in *xdata* space starting at address 8000h. The map file (page 9) shows that the linker has placed the *g_pVMI* array in *xdata* space from 8000h to 8003h as required.

2.3 Source File for PAL1

```
;PALASM Design Description
;---------------------------------------------- Declaration Segment ---------
TITLE FLASH/RAM CONTROL LOGIC
PATTERN A
REVISION 1.02                               ; FUSE CHECKSUM = 3EC4
AUTHOR MIKE TADYSHAK
COMPANY  TEXAS INSTRUMENTS, INC.
DATE   8/18/98
CHIP  PAL1 PAL16L8
;---------------------------------------------- PIN Declarations ------------
PIN  1    PSEN_                                          COMBINATORIAL ;
PIN  2    RD_                                            COMBINATORIAL ;
PIN  3    WR_                                            COMBINATORIAL ;
PIN  4    S1                                             COMBINATORIAL ;
PIN  5    S0                                             COMBINATORIAL ;
PIN  6    A15                                            COMBINATORIAL ;
PIN  10   GND                                                      ;
PIN  13   FRD_                                           COMBINATORIAL ;
PIN  14   FWR_                                           COMBINATORIAL ;
PIN  15   RRD_                                           COMBINATORIAL ;
PIN  16   RWR_                                           COMBINATORIAL ;
PIN  17   TCE_                                           COMBINATORIAL ;
PIN  18   TRD_                                           COMBINATORIAL ;
PIN  20   VCC                                                           ;
;---------------------------------------------- Boolean Equation Segment -----
EQUATIONS
  ; ENABLE ALL OUTPUTS
   FRD_.TRST   = VCC;
   FWR_.TRST   = VCC;
   RRD_.TRST   = VCC;
   RWR_.TRST   = VCC;
   TCE_.TRST   = VCC;
   TRD_.TRST   = VCC;
  ; STATE DEFINITIONS  (S1, S0)
  ;
  ; 1, 1 = EXECUTE FROM FLASH,    READ RAM  , WRITE RAM    (NORMAL)
  ; 1, 0 = EXECUTE FROM RAM  ,    READ RAM  ,  WRITE RAM   (LOADER)
  ; 0, 1 = EXECUTE FROM RAM  ,    READ RAM  ,  WRITE FLASH (PROGRAM)
  ; 0, 0 = EXECUTE FROM RAM  ,    READ FLASH               (POLL)
```

2.3 Source File for PAL1 (continued)

```
    ; FLASH read-enable
    ; Normal state: Flash is read by PSEN_ (code space)
    ; Poll state: Flash is read by RD_ (XDATA space) when polling
    ; for completion of the internal programming cycle
    /FRD_ =  S1 *  S0 * /PSEN_
          + /S1 * /S0 * /RD_;
    ;FLASH WRITE STROBE
    ; Flash program state: Write strobe from uC is routed to flash memory
    /FWR_ = /S1 *  S0 * /WR_;
    ;RAM read-enable
    ; Normal, Loader, Program states: Read control from uC is routed to RAM
    ; Loader, Program, Poll states: Code executes from RAM.
    /RRD_ =  S1    * /RD_
          +  S0    * /RD_
          + /S1    * /PSEN_
          + /S0    * /PSEN_;
    ;RAM WRITE STROBE
    ; Normal, Loader states: Write strobe from uC is routed to RAM.
    /RWR_ =  S1 * /WR_;
    ;VMI SLAVE CHIP ENABLE
    ; Normal state: Enabled for addresses in the 8000h to FFFFh range.
    /TCE_ =  S1 * S0 * A15;
    ;VMI SLAVE read-enable
    ; Normal state: Read control from uC is routed to VMI slave
    /TRD_ =  S1 * S0 * /RD_;
;----------------------------------- Simulation Segment ------------
SIMULATION
;-------------------------------------------------------------------
```

2.4 Source File: VMI_PORT.C

```
//
// VMI_Port.C
//
// Declaration for Memory–Mapped TVP5020 VMI Ports
// These are mapped to 8000–8003h by the make file APP_VMI.LIN
//
// The four VMI I/O locations of the TVP5020 are mapped as:
// g_pVMI[ADDRESS] accesses the VMI address register
// g_pVMI[DATA   ] accesses the VMI data    register
// g_pVMI[FIFO   ] accesses the VMI FIFO    register
// g_pVMI[STATUS ] accesses the VMI status  register

unsigned char xdata g_pVMI[4];
```

2.5 Make File: APP_VMI.LIN

```
NOLI RS(128) PL(68) PW(78)         &
XDATA( ?XD?VMI_PORT( 8000h ) )
```

2.6 Map File: APP_VMI.M51

```
BL51 BANKED LINKER/LOCATER V3.52                01/30/99  14:44:51  PAGE 1

MEMORY MODEL: LARGE

LINK MAP OF MODULE:  APP_VMI (MAIN)

            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            -----------------------------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     ”REG BANK 0”
                    0008H     0008H                  *** GAP ***
            REG     0010H     0008H     ABSOLUTE     ”REG BANK 2”
            IDATA   0018H     0001H      UNIT         ?STACK

            * * * * * * *  X D A T A   M E M O R Y  * * * * * * *
            XDATA   0000H     0005H     UNIT         ?XD?MAIN
            XDATA   0005H     0002H     UNIT         ?XD?TIMER
            XDATA   0007H     000EH     UNIT         ?XD?I2C
            XDATA   0015H     0004H     UNIT         ?XD?I2C6000
            XDATA   0019H     0002H     UNIT         ?XD?VMI5020
            XDATA   001BH     001DH     UNIT         _XDATA_GROUP_
                    0038H     7FC8H                  *** GAP ***
            XDATA   8000H     0004H     UNIT         ?XD?VMI_PORT

LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

2.7 Schematic Diagram—TVP5020 NTSC/PAL Video Decoder

The schematic diagram showing the TVP5020 and surrounding circuitry is shown in Figure 3. The VMI host interface includes eight data lines, two address lines, and five control lines.

- The bidirectional multiplexed address/data bus lines of the P80C652 is connected to the TVP5020 bidirectional data pins D7–D0 and are pulled up to VCC through 10 kΩ resistors.
- The least significant two address lines of the P80C652 (after demultiplexing) are connected to the TVP5020 address pins A1–A0 and are pulled up to VCC through 10 kΩ resistors.
- The TVP5020 VC0(RDY) output is left unconnected. Jumper JP12 must be removed.
- The $\overline{\text{RAMWR}}$ output from PAL1 must be connected to the TVP5020 VC1($\overline{\text{WR}}$) input. To make this connection, jumper JP6 across pins 2 and 3.
- The $\overline{\text{TRD}}$ output from PAL1 is connected directly to the TVP5020 VC2($\overline{\text{RD}}$) input.
- The $\overline{\text{TCE}}$ output from PAL1 must be connected to the TVP5020 VC3($\overline{\text{CE}}$) input. To make this connection, jumper JP5 across pins 1 and 2.
- The TVP5020 INTREQ pin is connected to the P80C652 user-defined I/O pin P3.2, and is read using the macro named INTREQ.

Figure 4 shows the TVP56000EVM board layout. Refer to this figure for the location and orientation of jumpers, switches, and connectors. The jumper settings are summarized in Table 3. The setting of JP4 is irrelevant, since the program does not make use of the serial port. Jumper settings for selection between CCIR601 and square pixel sampling rates are shown in Table 4.

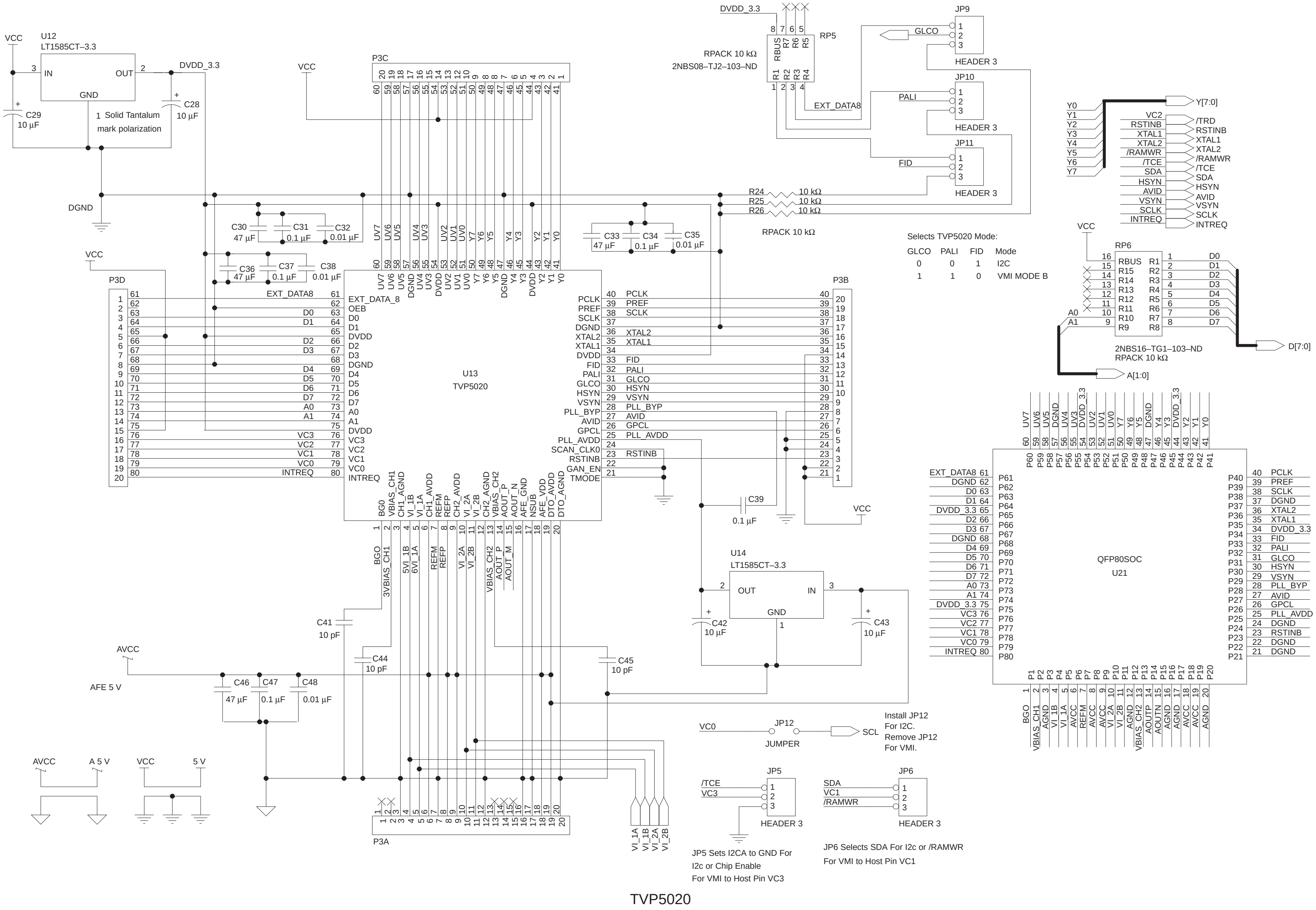


Figure 3. TVP5020 NTSC/PAL Video Decoder Schematic

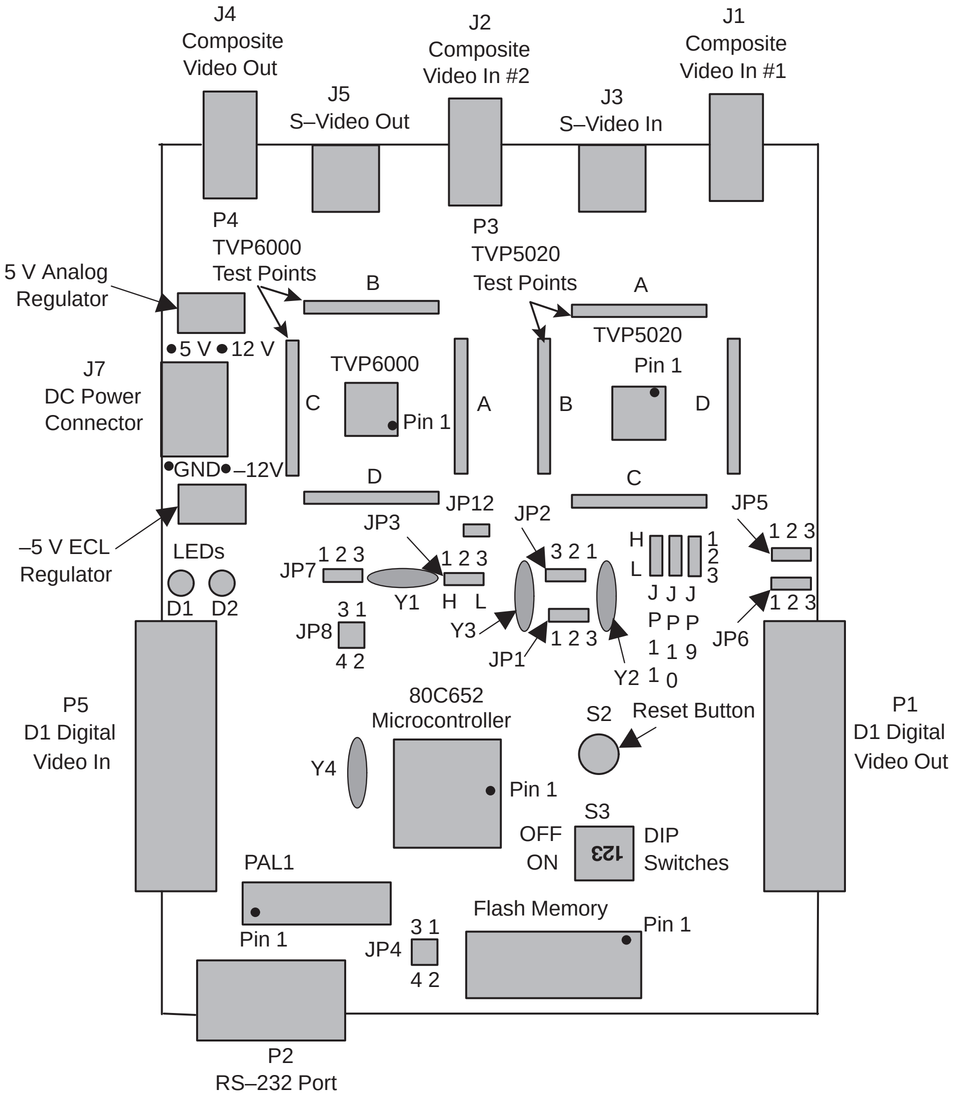


Figure 4. TVP56000EVM Board Layout

Table 3. Jumper Settings for TVP56000EVM Using VMI Bus

JUMPER(s)	JP3	JP4	JP7	JP8	JP5	JP6	JP9, JP10, JP11	JP12
Position	H	1–3, 2–4	1–2	1–2	1–2	2–3	H, H, L	OFF
Description	NORMAL	STRAIGHT CABLE	TVP6000: CLK RCVR	SCLK	VMI mode B			

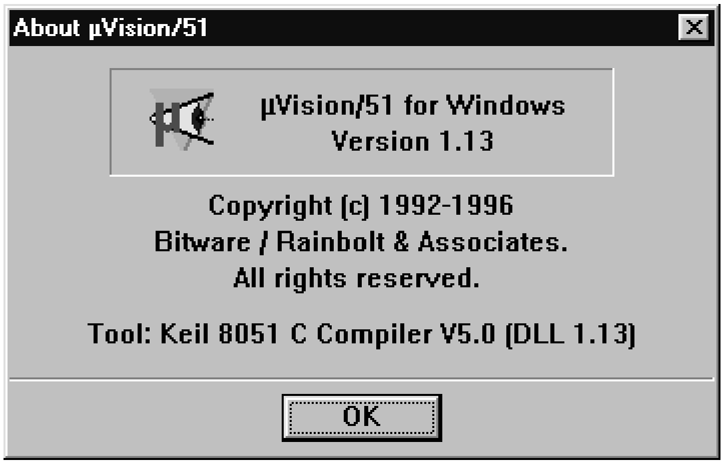
Table 4. Sampling Rate Dependent Jumper Settings

SAMPLING RATE	JP1	JP2
CCIR601	2–3	1–2
Square Pixel	1–2	2–3

3 Program Overview

This program is a complete working example of a C-language program to initialize the TVP5020 NTSC/PAL video decoder using the VMI bus interface. The program was compiled and linked using *uVision/51 for Windows,* a software package for compiling code for 80C51-type microcontrollers from Keil Software, Inc. Information about Keil Software can be found on the Internet at www.keil.com. See the Help-About dialog box for this software package (Figure 5). An in-circuit emulator for debugging P80C652 code is the USP-51 from Signum Systems. Information about Signum Systems can be found on the Internet at www.signum.com. See the Help-About dialog box for the USP-51 emulator software (Figure 6).

The linker output is in standard Intel Intellec 8/MDS format. This format is widely supported by PROM programmers. The program can then be installed on the TVP56000EVM by programming the flash memory device (Atmel 29C512) using a PROM programmer. The program contains TVP5020 microcode for four video modes to enable testing of NTSC and PAL video standards using CCIR601 and square pixel sampling rates. The video mode is selected by setting the DIP switches as was shown in Table 2.



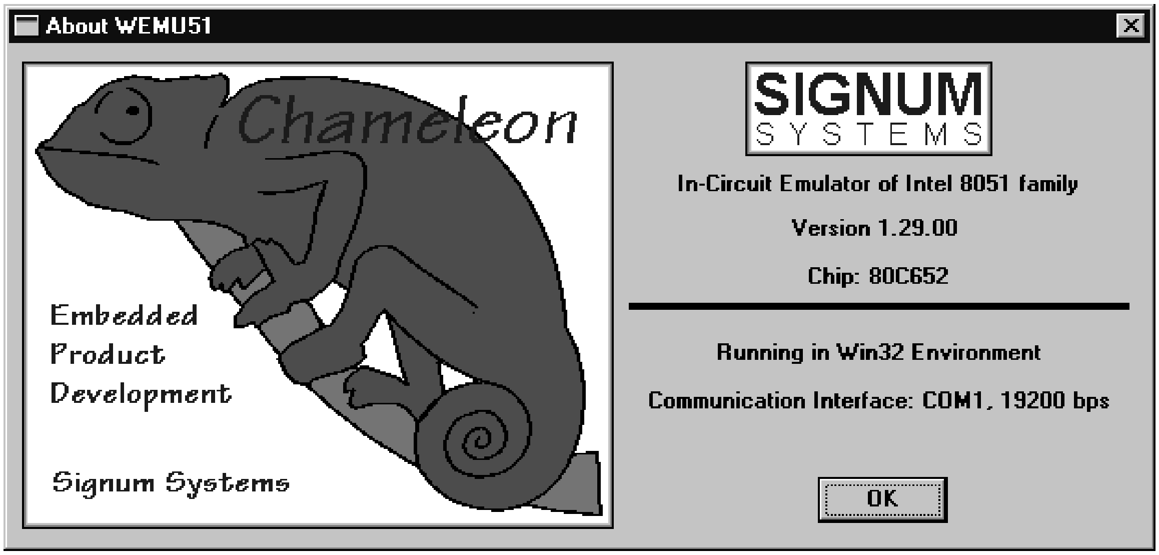
About µVision/51

µVision/51 for Windows
Version 1.13

Copyright (c) 1992-1996
Bitware / Rainbolt & Associates.
All rights reserved.

Tool: Keil 8051 C Compiler V5.0 (DLL 1.13)

Figure 5. Help-About Dialog Box from uVision/51 for Windows



About WEMU51

In-Circuit Emulator of Intel 8051 family

Version 1.29.00

Chip: 80C652

Running in Win32 Environment

Communication Interface: COM1, 19200 bps

Figure 6. Help-About Dialog Box Signum Systems In-Circuit Emulator

3.1 Microcontroller-Specific Macros

Most of the source code is in standard C-language. The main exception is the use of macros to access the special-function registers or special-function I/O bits as defined for the P80C652. The complete set of macros defined for the P80C652 is contained in the file REG652.H, which is shown in section 3.2. As Table 1 shows, there are several user-definable I/O pins assigned for special use by the EVM. These are named in the P1 and P3 I/O ports sections of the header file. Of these user-definable I/O pins, this program uses only INTREQ, SW2, SW3, and T6RESET. SW2 and SW3 are used to read the corresponding DIP switches, and T6RESET provides a software reset for the TVP6000 video encoder.

Many of the P80C652-specific special-function registers are used to control the on-chip general-purpose timer and the I^{2}C Bus interface. These are localized to the TIMER and I^{2}C source-code modules.

The P80C652 provides 128 bytes of on-chip RAM, and direct support for 64 kB of external data memory (*xdata* space), and 64 kB of read-only memory (*code* space). The keywords *xdata* and *code* are sometimes required in variable declarations to specify the type of memory storage. For example, the arrays of *unsigned char,* which hold the TVP5020 microcode modules, are declared with the *code* keyword so that they get stored in flash memory.

3.2 Header File: REG652.H

```
// REG652.H
// Header file for Philips P80C652 Microcontroller.

/*  BYTE Registers */
sfr P0     = 0x80;
sfr P1     = 0x90;
sfr P2     = 0xA0;
sfr P3     = 0xB0;

sfr PSW    = 0xD0;
sfr ACC    = 0xE0;
sfr B      = 0xF0;
sfr SP     = 0x81;
sfr DPL    = 0x82;
sfr DPH    = 0x83;
sfr PCON   = 0x87;
sfr TCON   = 0x88;
sfr TMOD   = 0x89;
sfr TL0    = 0x8A;
sfr TL1    = 0x8B;
sfr TH0    = 0x8C;
sfr TH1    = 0x8D;
sfr IE     = 0xA8;
sfr IP     = 0xB8;
sfr S0CON  = 0x98; /* UART control */
sfr S0BUF  = 0x99; /* UART data buffer */

sfr S1CON  = 0xD8; /* I2C control register */
sfr S1STA  = 0xD9; /* I2C status register */
sfr S1DAT  = 0xDA; /* I2C data register */
sfr S1ADR  = 0xDB; /* I2C address register */


/*  BIT Register  */
/*  PSW  */
sbit CY    = 0xD7;
sbit AC    = 0xD6;
sbit F0    = 0xD5;
sbit RS1   = 0xD4;
sbit RS0   = 0xD3;
sbit OV    = 0xD2;
sbit P     = 0xD0;

/*  TCON */
sbit TF1   = 0x8F;
sbit TR1   = 0x8E;
sbit TF0   = 0x8D;
sbit TR0   = 0x8C;
sbit IE1   = 0x8B;
sbit IT1   = 0x8A;
sbit IE0   = 0x89;
sbit IT0   = 0x88;
```

3.2 Header File: REG652.H (continued)

```
/*  IE  */
sbit EA    = 0xAF;
sbit ES1   = 0xAD; /* I2C interrupt enable */
sbit ES0   = 0xAC; /* UART interrupt enable */
sbit ET1   = 0xAB;
sbit EX1   = 0xAA;
sbit ET0   = 0xA9;
sbit EX0   = 0xA8;

/*  IP */
sbit PS1   = 0xBD;
sbit PS0   = 0xBC;
sbit PT1   = 0xBB;
sbit PX1   = 0xBA;
sbit PT0   = 0xB9;
sbit PX0   = 0xB8;

//  P1
sbit SDA      = 0x97;
sbit SCL      = 0x96;
sbit T6RESET  = 0x95;
sbit S1       = 0x94;
sbit SW3      = 0x93;
sbit SW2      = 0x92;
sbit SW1      = 0x91;
sbit S0       = 0x90;

//  P3
sbit RD       = 0xB7;
sbit WR       = 0xB6;
sbit LED2     = 0xB5;
sbit LED1     = 0xB4;
sbit FLASHJMP = 0xB3;
sbit INTREQ   = 0xB2;
sbit TXD      = 0xB1;
sbit RXD      = 0xB0;

/*  S0CON  */
sbit SM1   = 0x9E;
sbit SM2   = 0x9D;
sbit REN   = 0x9C;
sbit TB8   = 0x9B;
sbit RB8   = 0x9A;
sbit TI    = 0x99;
sbit RI    = 0x98;

/*  S1CON   */
sbit CR0   = 0xD8;
sbit CR1   = 0xD9;
sbit AA    = 0xDA;
sbit SI    = 0xDB;
sbit STO   = 0xDC;
sbit STA   = 0xDD;
sbit ENS1  = 0xDE;
sbit CR2   = 0xDF;
```

3.3 Source-Code Modules

Table 5 summarizes the relationships between the various source-code modules. Each source-code module is contained in one *.C* source file and has an associated .H header file. The *Timer* and *I2C* modules are described as microcontroller-specific. In order to port these functions to another hardware environment, equivalent functions, written for the specific processor, would need to be supplied or created. The *Main* and *I2C6000* modules could be used virtually unchanged. In the new environment, the TVP6000 software reset (T6RESET), as well as the reading of the INTREQ and DIP-switch lines SW2 and SW3, would have to be implemented.

Table 5. Source Code Module Relationships

SOURCE-CODE MODULE	DESCRIPTION	MICROCONTROLLER-SPECIFIC?	TVP56000EVM-SPECIFIC?	CALLS FUNCTIONS IN THESE MODULES
Main	Main program	No	Yes (uses T6RESET)	I2C5020, I2C6000, Timer
Timer	General-purpose timer routines and ISR	Yes	No	None
I2C	I2C bus routines and ISR	Yes	No	Timer
VMI5020	TVP5020 VMI routines	No	Yes (uses INTREQ, SW2 and SW3)	None
I2C6000	TVP6000–specific I2C routines	No	No	I2C, Timer

4 Program Description

4.1 Source-Code Module: Main

4.1.1 Inclusion of TVP5020 Microcode Files (Lines 11–14)

Header files containing the TVP5020 microcode are included. These provide support for NTSC and PAL video standards with CCIR601 or square pixel sampling. Each header file declares an array of type *unsigned char*. The first byte of each array is the subaddress for writing to the TVP5020 program memory (0x7E). The TVP5020 microcode is supplied in a five-character Hex-ASCII format (Figure 7). Conversion to a standard C-language header file can be done with a utility called HexConv. The output of HexConv is shown in Figure 8. If necessary, the *#define* constant for the code size (which includes the subaddress byte – 0x7E) as well as the array name may be given a unique name. If the target processor is an 80C51 derivative, the keyword *code* must be inserted. Also, adding a comment identifying the microcode type and version is very helpful. The resulting microcode file is shown in Figure 9.

4.1.2 Function: Main()

4.1.2.1 Declaration of TVP5020 Register Patch Data (Lines 16–24)

After the microcode is downloaded and the TVP5020 CPU is restarted, the registers are initialized with their default values (as defined by the TVP5020 data manual) by the internal CPU. Some registers must be *patched* with a different value for the TVP5020 to function properly on the TVP56000EVM. The array *g_pTVP5020Patch[ ]* contains the address and data for three registers that must be modified. Table 6 describes these register changes.

Table 6. TVP5020 Register Patches

ADDRESS	DEFAULT DATA	PATCHED DATA	COMMENT
03h	00h	19h	Enable HSYN, VSYN, AVID, SCLK, PCLK and YUV outputs
07h	00h	10h	Bypass luminance processing during vertical blank
0Dh	00h	0Fh	Select 8-bit ITU-R BT.656 interface

```
80000
00000
303FC
.
.
.
C3F80
```

Figure 7. TVP5020 Microcode in Hex-ASCII Format

```
#define TVP5020_CODE_SIZE 0x27b8

unsigned char TVP5020_CODE[] =
{
0x7E,
0x08,
0x00,
0x00,
0x00,
0x00,
0x03,
0x03,
0xFC,
.
.
.
0x0C,
0x3F,
0x80
};
```

Figure 8. TVP5020 Microcode after Conversion to Standard C Format

```
#define TVP5020_N601_CODE_SIZE 0x27b8

// TVP5020 NTSC CCIR601 Version: 63

unsigned char code T520_N601[] =
{
0x7E,
0x08,
0x00,
0x00,
0x00,
0x00,
0x00,
0x03,
0x03,
0xFC,
.
.
.
0x0C,
0x3F,
0x80
};
```

Figure 9. TVP5020 Microcode after Modification

4.1.2.2 Initialization (Lines 37–45)

The function call *timer0_initialize()* initializes the P80C652 on-chip general-purpose timer to generate a timer-tick interrupt every 2 ms. Next, *timer0_wait()* is called to produce a 100 ms delay to insure stabilization after reset. The call to *DecoderReset()* configures the TVP5020 interrupt request output pin (INTREQ).

4.1.2.3 Video Mode Selection (Lines 47–72)

The current state of the DIP switches is read. The two lsbs are used to select the video mode as shown in Table 2. The global variables *g_nROMCodeSize* and *g_pROMCode* are initialized with the size of the selected microcode file (in bytes and including the subaddress byte) and with the starting address of the selected microcode data array.

4.1.2.4 Reset Timer-Tick (Line 75)

The call to *ResetTickCount()* resets the internal count of timer-tick interrupts (which occur every 2 ms). The timer-tick count can run up to about 128 seconds before rolling over. In a program with multiple uses for the general-purpose timer, the timer-tick count should be reset only in the outermost loop.

4.1.2.5 Power-up Initialization (Line 78)

The call to *PowerUpInitialization()* performs the tasks of downloading the TVP5020 microcode, restarting the TVP5020 internal CPU, patching the TVP5020 registers, and initializing the TVP6000 video encoder. One parameter is passed to *PowerUpInitialization()* to indicate the selected video mode. Upon return from *PowerUpInitialization()*, the program spins in an endless loop.

4.1.3 Function: Power-up Initialization()

4.1.3.1 Microcode Download (Line 94)

The call to *HandleDownload()* calls the specific routine which will download the microcode to the video decoder. The code size and code pointer variables are passed as parameters. *HandleDownload()* routines have been written for the I2C, VIP, and VMI interfaces. The source-code module *VMI5020.C* contains the version of *HandleDownload()* for the VMI Bus.

4.1.3.2 Restart Microprocessor (Lines 96–102)

After the microcode download completes, the internal microprocessor is restarted. This is done by writing a 00h byte (the data can be any value) to the restart subaddress (7Fh). The function *WriteTVP5020()* is used whenever it is required to write to a TVP5020 register. The parameters passed to *WriteTVP5020()* are a byte count (this is always 2) and a pointer to the storage location of the subaddress and data byte. A 10 ms wait is inserted after the restart command to enable the internal microprocessor to complete its initialization code.

4.1.3.3 Patch TVP5020 Registers (Lines 104–105)

The call to PatchTVP5020Registers() implements the register modifications which were described in Section 4.1.2.1 and Table 6.

4.1.3.4 Reset the TVP6000 Video Encoder (Lines 107–110)

The user-definable I/O pin P1.5 is used as a software-controlled reset for the TVP6000. T6RESET is a macro which allows control of pin P1.5. The TVP6000 reset input is held active for 100 ms after the TVP5020 is initialized. This is needed, since the TVP6000 is not guaranteed to have received a clock from the TVP5020 during power-up reset.

4.1.3.5 Initialize the TVP6000 Video Encoder (Lines 112–113)

The call to *LoadTVP6000()* initializes the video encoder registers. One parameter is passed to indicate the selected video mode. The register data is located in the header file *DATA6000.H*.

4.1.4 Function: Patch TVP5020 Registers()

4.1.4.1 Loading the Registers (Lines 120–123)

A *for* loop is used to patch the TVP5020 registers using the data in the *g_pTVP5020Patch* array. This array is a global variable and was initialized in lines 17–24. The constant *TVP5020_PATCH_SIZE* holds the number of bytes in the array and must be changed if the number of register writes is changed.

4.2 Header File: Main.H

```
// Main.H
//
// Header file for main program to initialize the TVP5020 Video Decoder
//
#define FALSE                   0
#define TRUE                    !FALSE

#define TVP5020_RESTART_SUB_ADDR        0x7F

void             main (void);
void             DecoderReset( void );
unsigned char    ReadSwitch( void );
void             PowerUpInitialization( int nSwitch );
void             HandleDownload( unsigned nCount, unsigned char* pInBuf );
unsigned         WriteTVP5020(int nLength, unsigned char *pBuf );
void             PatchTVP5020Registers(void);
```

4.3 Source File: Main.C

```
DOS C51 COMPILER V5.10, COMPILATION OF MODULE MAIN
OBJECT MODULE PLACED IN MAIN.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE MAIN.C DB OE OR LARGE

stmt level       source

   1             // Main.C
   2             //
   3             // Main program to initialize the TVP5020 Video Decoder
   4             //
   5             #include ”Main.h”
   6             #include ”Timer.h”
   7             #include ”Reg652.h”
   8             #include ”I2C6000.H”
   9
  10             //TVP5020 microcode files
  11             #include ”5020NSQP.H”
  12             #include ”5020N601.H”
  13             #include ”5020PSQP.H”
  14             #include ”5020P601.H”
  15
  16             // Registers to modify after TVP5020 CPU startup
  17             #define      TVP5020_PATCH_SIZE         6
  18             unsigned char code g_pTVP5020Patch[] =
  19             {
  20                // subaddress, data
  21                0x03, 0x19,
  22                0x07, 0x10,
  23                0x0D, 0x0F
  24             };
  25
  26             // Size of TVP5020 microcode file (defined in 5020xxxx.H)
  27             unsigned        g_nROMCodeSize   = TVP5020_N601_CODE_SIZE;
  28
  29             // Pointer to the TVP5020 microcode
  30             unsigned char*   g_pROMCode      = T520_N601;
  31
```

4.3 Source File: Main.C (Continued)

```
       void main(void)
        {
1           // DIP Switch value
1           unsigned char nSwitch = 0;
1
1           // Initialize general purpose timer
1           timer0_initialize();
1
1           /* Wait 100ms – for stabilization after reset */
1           timer0_wait(ONE_HUNDRED_MS);
1
1           // For VMI Bus, this configures the TVP5020 interrupt output
            //(INTREQ)
1           // For VIP Bus, this sends a reset code to the VIP emulation FPGA
1           DecoderReset();
1
1           // Two LSBs of switch select the video mode
1           nSwitch = ReadSwitch() & 3;
1
1           // Point to the microcode selected by the DIP switch
1           switch( nSwitch )
1           {
2                 case 0:
2                      g_pROMCode      = T520_N601;
2                      g_nROMCodeSize  = TVP5020_N601_CODE_SIZE;
2                  break;
2
2                  case 1:
2                      g_pROMCode      = T520_NSQP;
2                      g_nROMCodeSize  = TVP5020_NSQP_CODE_SIZE;
2                  break;
2
2                  case 2:
2                      g_pROMCode      = T520_P601;
2                      g_nROMCodeSize  = TVP5020_P601_CODE_SIZE;
2                  break;
2
2                  case 3:
2                      g_pROMCode      = T520_PSQP;
2                      g_nROMCodeSize  = TVP5020_PSQP_CODE_SIZE;
2                  break;
2           }
1
1           // Reset timer-tick to avoid rollover
1           ResetTickCount();
1
1           // Initialize the video mode
1           PowerUpInitialization( nSwitch );
1
1           // After video is initialized, do nothing
1           while(1)
1           {
2           ;
2           }
1
1           return;
1      }
```

4.3 Source File: Main.C (Continued)

```

              void PowerUpInitialization( int nSwitch )
              {
       1          unsigned char buf[2];
       1
       1          /* Download video decoder microcode */
       1          HandleDownload( g_nROMCodeSize, g_pROMCode );
       1
       1          //Restart microprocessor
       1          buf[0] = TVP5020_RESTART_SUB_ADDR;
       1          buf[1] = 0;
       1          WriteTVP5020( 2, buf );
       1
       1          // Wait 10ms for TVP5020 CPU to start–up
       1          timer0_wait(TEN_MS);
       1
       1          // Modify registers from the default state as required
       1          PatchTVP5020Registers();
       1
       1          // Reset the TVP6000
       1          T6RESET = 0;
       1          timer0_wait(ONE_HUNDRED_MS);
       1          T6RESET = 1;
       1
       1          // Initialize TVP6000
       1          LoadTVP6000( nSwitch );
       1      }

              void PatchTVP5020Registers (void)
              {
       1          int i = 0;
       1
       1          for( i=0; i<TVP5020_PATCH_SIZE; i+=2 )
       1          {
       2              WriteTVP5020( 2, g_pTVP5020Patch+i );
       2          }
       1
       1          return;
       1      }


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

4.4 Source-Code Module: I2C

This source-code module is included only for communication with the TVP6000 NTSC/PAL video encoder.

4.4.1 Function: initia_i2c() (Lines 28–41)

This function initializes the I2C bus signals (SCL and SDA) to a high level. The internal P80C652 I2C interrupt is enabled and given low (normal) priority. The P80C652 on-chip I2C controller is initialized to be the I2C bus master with a baud rate of 92.16 kHz. This frequency is the P80C652 oscillator frequency (11.0592 MHz) divided by 120.

4.4.2 Function: start_i2c()

This function is called to perform a transaction on the I2C bus.

4.4.2.1 Initialize Variables for the ISR (Lines 54–64)

The current timer-tick count is saved to be used later to determine if a timeout condition has occurred. The macro *EA* is used to globally enable or disable all interrupts. The global variables used by the I2C interrupt service routine (ISR) are:

- **b_counter**—Byte counter. Initialized to 00h, counts up to the terminal count value.
- **num_b**—Number of bytes. Holds the terminal count value.
- **slave_rw**—Slave device ID and read/write bit.
- **i2cbuf**—Pointer to caller’s data buffer.

4.4.2.2 Start the I2C Transaction (Lines 63–64)

The macro *STA* is used to set the start bit in the I2C control register. Then, the global interrupt control bit *EA* enables the hardware interrupts.

4.4.2.3 Wait for the I2C Bus Transaction to Complete (Lines 66–73)

The program now remains in a loop waiting for either all bytes to be transferred or the occurrence of an error condition. Meanwhile, I2C bus interrupts are occurring and the I2C ISR is controlling the data transfer. The timer-tick count is checked for a timeout condition by comparing the elapsed time with *g_nI2Ctimeout*. The value in *g_nI2Ctimeout* is in units of timer-ticks. The timer-tick is programmed to occur once every 2 ms.

4.4.3 Function: i2c_isr() (Lines 77–266)

This is the interrupt service routine (ISR) for the I2C bus interface. The I2C controller is embedded in the P80C652. A simple register interface provides access to address, data, control and status registers. Each time an I2C interrupt occurs, the status register (S1STA) is read to obtain the current state code from the I2C controller. The state code is used to branch to the appropriate code segment to handle the interrupt. The I2C global variables are updated and data is transferred to/from the user’s data buffer. The states for the master transmitter and master receiver are described in Tables 7 and 8. The last step of interrupt handling is writing one of the following four codes back to the I2C control register (S1CON) to request a specific action:

- I2C_RLS_STA —Release bus and generate a start condition
- I2C_RLS_ACK —Release bus and acknowledge the data transfer
- I2C_RLS_NACK —Release bus and do not acknowledge the data transfer
- I2C_STOP —Generate a stop condition

Table 7. I2C Controller: Master Transmitter States

I2C CONTROLLER STATE	DEFINITION	NEXT ACTION TAKEN BY I2C ISR
MASTER TRANSMITTER STATES		
08h	Start condition has been transmitted	Send slave address + r/w bit
10h	Repeat start condition has been transmitted	Send slave address + r/w bit
18h	Slave address has been sent and ACK was received	Transmit first data byte
20h	Slave address has been sent and NOT ACK was received	Transmit first data byte. Flag I2C NOT ACK error.
28h	Data has been transmitted and ACK has been received	Transmit next data byte. If all data has been transmitted, issue a stop condition.
30h	Data has been transmitted and NOT ACK has been received	Transmit next data byte. If all data has been transmitted, stop the bus. Flag I2C NOT ACK error.
38h	Bus arbitration lost	Flag I2C bus arbitration lost error. Issue another start condition.

Table 8. Controller: Master Receiver States

I2C CONTROLLER STATE	DEFINITION	NEXT ACTION TAKEN BY I2C ISR
MASTER RECEIVER STATES		
40h	Slave address has been sent and ACK was received	If transaction involves only one data byte, signal the controller to NOT ACK the next data byte received. Otherwise, signal the controller to acknowledge the next data byte received.
48h	Slave address has been sent and NOT ACK was received	If transaction involves only one data byte, signal the controller to NOT ACK the next data byte received. Otherwise, signal the controller to acknowledge the next data byte received. Flag I2C NOT ACK error.
50h	Data has been received and ACK has been transmitted	If this is previous to the NEXT-TO-LAST data byte, signal the controller to acknowledge the next data byte received. If this is the next-to-last data byte, signal the controller to NOT ACK the next data byte received.
58h	Data has been received and NOT ACK has been transmitted	If this is previous to the LAST data byte, signal the controller to acknowledge the next data byte received and flag a I2C NOT ACK error. If this is the last data byte, then issue a stop condition.
F8h	No relevant state information is available	No action required
00h	I2C bus error due to detection of an illegal start or stop condition, or I2C controller detected entry into an illegal state.	Flag an I2C bus error
Other	I2C controller reported a state which is not supported by the interrupt service routine	Flag an I2C unsupported state error

4.5 Header File: 12C.H

```
// I2C.H
//
// Header file for I2C bus routines
//
#define FALSE                   0
#define TRUE                   !FALSE

#define ERR_I2C_NOTACK     0x01
#define ERR_I2C_ARBILOST   0x02
#define ERR_I2C_GERROR     0x04
#define ERR_I2C_TIMEOUT    0x08
#define ERR_I2C_BUSERROR   0x10
#define ERR_I2C_DEVID      0x20

void     initia_i2c(void);
unsigned start_i2c(unsigned char i2c_addrs, int buf_length,
                unsigned char bufaddrs);
```

4.6 Source File: I2C.C

```
DOS C51 COMPILER V5.10, COMPILATION OF MODULE I2C
OBJECT MODULE PLACED IN I2C.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE I2C.C DB OE OR LARGE

stmt level  source

  1          //
  2          // I2C.C
  3          //
  4          // Routines for I2C Bus
  5          //
  6          #include ”I2C.h”
  7          #include ”Timer.h”
  8          #include ”reg652.h”
  9
 10          #define FALSE           0
 11          #define TRUE            !FALSE
 12          #define I2C_STOP        0xD5      /* generated a STOP condition on I2C, and
                                               100 kBps */
 13          #define I2C_RLS_ACK     0xC5      /* Release bus and generate a ACK */
 14          #define I2C_RLS_NACK    0xC1      /* Release bus and generate a NOT ACK */
 15          #define I2C_RLS_STA     0xE5      /* Release bus and generate START */
 16
 17          // I2C Timeout
 18          unsigned       g_nI2CTimeout   = TEN_SECONDS;
 19
 20
 21          unsigned xdata error_i2c = 0;     /* I2C Errors */
 22          unsigned xdata b_counter = 0;     /* length of i2c send buffer */
 23          unsigned xdata num_b     = 0;     /* number of bytes that will be sent/read */
 24          unsigned xdata num_b_minus_1 = 0; /* number of bytes that will be
 25                                               sent/read – 1 */
 25          unsigned char xdata slave_rw = 0; /* slave address plus read/write direction */
 26          static unsigned char *i2cbuf = (unsigned char*)0; /* pointer to I2C
 27                                               send/receiving buffer */
 27
 28          /*
 29          ---------------------------------------------------------------------
 30          This function will initialize the I2C interface
 31          ---------------------------------------------------------------------
 32          */
 33
 34          void initia_i2c(void)
 35          {
 36     1       SDA = 1; /* set data pin as high level */
 37     1       SCL = 1; /* set clock pin as high level */
 38     1       ES1 = 1; /* enable I2C interrupt */
 39     1       PS1 = 0; /* set I2C interrupt PRIORITY level as LOW */
 40     1       S1CON = I2C_RLS_ACK; /* set 80C652 as a master only, bit rate = 92.16k */
 41     1    }
```

4.6 Source File: I2C.C (continued)

```

       /*
       ---------------------------------------------------------------------
       This function transfers one block of data in or out
       ---------------------------------------------------------------------
       */

         unsigned start_i2c( unsigned char i2c_addrs, int buf_length,
                             unsigned char *bufaddrs )
       {
1        unsigned xdata start_point;
1        unsigned test = 0u;
1
1        // Set a reference time
1        start_point = current_tick ();
1
1        EA = 0;             // Disable i2c interrupt
1        b_counter = 0;
1        num_b = (unsigned)buf_length;
1        num_b_minus_1 = num_b – 1;
1        slave_rw = i2c_addrs;
1        i2cbuf = bufaddrs;  // initialized the buffer point
1        STA = 1;            // set STA bit of S1CON, start I2C
1        EA = 1;
1
         /* wait until all data in buffer have been sent out */
1        while ( (b_counter < num_b) && ( error_i2c == 0u) )
1         {
2                  if (timer0_elapsed_count(start_point) > g_nI2Ctimeout)
2                  {
3                       error_i2c |= ERR_I2C_TIMEOUT;
3                  }
2         }
1        return( b_counter );
1      }

       /*
       ---------------------------------------------------------------------
        I2C interrupt service routine
        interrupt number=5, address=0x002Bh, using register bank2
       ---------------------------------------------------------------------
       */

       static void i2c_isr (void) interrupt 5 using 2
       {
1
1      unsigned char i2cst;
1      unsigned char nDummy = 0;
1      i2cst = S1STA;
1
```

4.6 Source File: I2C.C (continued)

```
1         switch (i2cst)
1         {
2            /*------------------------------------------------*/
2            /* following section will be MASTER transmit mode */
2            /*------------------------------------------------*/
2
2            /* a START condition has been sent */
2            /* will send out slave address + r/w bit */
2            case 0x08:
2                S1DAT = slave_rw;
2                S1CON = I2C_RLS_ACK;
2            break;
2
2            /* a repeat START has been transmitted */
2            /* will load SLA+R/W, and return ACK */
2            case 0x10:
2            S1DAT = slave_rw;
2            S1CON = I2C_RLS_ACK;
2            break;
2
2            /* slave address has been send and ACK received */
2            /* will send out 1st byte of data */
2            case 0x18:
2            S1DAT = *i2cbuf; /* load a byte to data register */
2            S1CON = I2C_RLS_ACK;
2            break;
2
2            /* NOT ACK received, will send out 1st byte of data anyway */
2            case 0x20:
2            S1DAT = *i2cbuf; /* load a byte to data register */
2            S1CON = I2C_RLS_ACK;
2            error_i2c |= ERR_I2C_NOTACK;
2            break;
2
2            /* continue sending data */
2            /* 1st byte of data has been sent and ACK received */
2            /* If all the data were sent, then transmit a STOP */
2            /* else continue to transmit next byte */
2            case 0x28:
2                   b_counter++;
2                   // Last state of b_counter will be num_b
2                   if ( b_counter < num_b )
2                   {
3                   S1DAT = *(i2cbuf+b_counter); /* send 1 byte data */
3
3                   //
3                   S1CON = I2C_RLS_ACK;
3                   }
2                   else
2                   {
3                      S1CON = I2C_STOP; /* all data were sent,stop bus */
3                   }
2            break;
```

4.6 Source File: I2C.C (continued)

```
2
2               /* 1st byte of data has been sent but NOT ACK rcvd */
2           case 0x30:
2                  b_counter++;
2                  // Last state of b_counter will be num_b
2                  if ( b_counter < num_b )
2                  {
3                  S1DAT = *(i2cbuf+b_counter); /* send 1 byte data */
3                  S1CON = I2C_RLS_ACK;
3                  }
2                  else
2                  {
3                  S1CON = I2C_STOP;   /* all data were sent, stop bus */
3                  }
2
2                  error_i2c |= ERR_I2C_NOTACK;
2           break;
2
2           /* Bus arbitration lost, release bus and try to restart */
2           case 0x38:
2               S1CON = I2C_RLS_STA;
2              error_i2c |= ERR_I2C_ARBILOST;
2           break;
2
2           /*-----------------------------------------------*/
2           /* following section will be MASTER receive mode */
2           /*-----------------------------------------------*/
2
2           /*SLA+R has been sent, ACK received */
2           case 0x40:
2           if( num_b == 1 )
2           {
3              // Only one byte will be received, don’t acknowledge
3              // This will signal the slave transmitter to stop
3              S1CON = I2C_RLS_NACK;
3           }
2           else
2           {
3              // More than one byte will be received, acknowledge
               // the first one
3              S1CON = I2C_RLS_ACK;
3           }
2
2           break;
2
2
```

4.6 Source File: I2C.C (Continued)

```
2            /* NOT ACK received on SLA+R, will ignore it */
2            case 0x48:
2            if( num_b == 1 )
2            {
3                // Only one byte will be received, don’t acknowledge
3                // This will signal the slave transmitter to stop
3                S1CON = I2C_RLS_NACK;
3            }
2            else
2            {
3                // More than one byte will be received, acknowledge
                 // the first one
3                S1CON = I2C_RLS_ACK;
3            }
2            error_i2c |= ERR_I2C_NOTACK;
2            break;
2
2
2            /* a byte has been received, and ACK was returned */
2            case 0x50:
2                /* read one byte from S1DAT */
2                *(i2cbuf + b_counter) = S1DAT;
2                b_counter++;
2
2                // if this is prior to the next–to–last byte
2                if ( b_counter < num_b_minus_1 )
2                {
3                    // Acknowledge the next byte received
3                    S1CON = I2C_RLS_ACK;
3                }
2                // if this is the next–to–last byte
2                else
2                {
3                    // Do not acknowledge the next byte received
                     //  (the last byte)
3                    // This will signal the slave transmitter to stop
3                    S1CON = I2C_RLS_NACK;
3                }
2            break;
2
2
```

4.6 Source File: I2C.C (continued)

```
2           // a byte received and NOT ACK was returned
2           // This should be the last byte received
2           case 0x58:
2              /* read one byte from S1DAT */
2              *(i2cbuf + b_counter) = S1DAT;
2              b_counter++;
2
2              /* if this is not the last byte – error condition */
2              if ( b_counter < num_b )
2              {
3                     S1CON = I2C_RLS_ACK;
3                     error_i2c |= ERR_I2C_NOTACK;
3              }
2              /* if this is the last byte, then STOP bus */
2              else
2              {
3                     S1CON = I2C_STOP;
3              }
2           break;
2
2           // No Relevant state information is available
            // no action required
2           case 0xF8:
2              nDummy = 0;
2           break;
2
2           // Bus error due to illegal start or stop condition
            // or SIO1 has entered an illegal state
2           case 0x00:
2                     S1CON = I2C_RLS_ACK;
2                     error_i2c |= ERR_I2C_BUSERROR;
2           break;
2
2           /* all other cases will be error in this system */
2           default:
2                     S1CON = I2C_RLS_ACK;
2                     error_i2c |= ERR_I2C_GERROR;
2           break;
2        }
1      }

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

4.7 Source-Code Module: Timer

The general-purpose timer is used to insert time delays and to determine when a timeout condition has occurred. The timer is programmed so that a timer ‘tick’ interrupt occurs every 2 ms.

4.7.1 Function: timer0_isr() (Lines 26–59)

This is the ISR for the general-purpose timer. The timer is stopped. A constant is loaded into the timer data registers (TL0 and TH0). The timer is restarted. The timer increments and generates an interrupt when it reaches its maximum count. Each time the timer-tick interrupt occurs, the global variable ‘timer0_tick’ is incremented by 1. The constant was calculated so that the time from the timer restart until it reaches its maximum count is 2 ms. The equation for calculating the timer reload value (TH0, TL0) from the desired timer-tick period (T) is shown below. The calculated timer reload value with the 11.0592 MHz crystal and a timer-tick period of 2 ms is F8CDh.

$\text{TH0, TL0} = 10000\text{h} - ((f_{osc\ /}\ 12) * T)$

$\text{TH0, TL0} = 10000\text{h} - ((11059200 / 12) * 0.002)$

$\text{TH0, TL0} = \text{F8h, CDh}$

4.7.2 Function: timer0_initialize() (Lines 61–94)

This function initializes the general-purpose timer. It is called once at program startup. The timer-tick count is initialized to zero and then the timer is stopped. The timer mode is set for 16-bit counter with no prescaling, and then the timer reload value is written. The timer interrupt is enabled and given low (normal) priority, and then the timer is restarted. The global interrupt control (EA) is enabled. At this point, the timer-tick interrupts start occurring.

4.7.3 Function: ResetTickCount() (Lines 96–112)

The call to *ResetTickCount()* resets the timer-tick count to zero. The timer-tick count can run up to about 128 seconds before rolling over. In a program with multiple uses for the general-purpose timer, the timer-tick count should be reset only in the outermost loop.

4.7.4 Function: current_tick() (Lines 114–131)

This function returns the current timer-tick count.

4.7.5 Function: timer0_elapsed_count() (Lines 133–150)

This function returns the number of elapsed timer-tick counts. The parameter is the starting timer-tick count from which to measure the elapsed time.

4.7.6 Function: timer0_wait() (Lines 152–167)

This function generates a time delay. The parameter is the number of timer-tick counts to delay.

4.8 Header File: Timer.H

```
// Timer.H
//
// Header file for P80C652 microcontroller general purpose timer routines
//
#define TCLK           11059200       /* Clock speed in Hz */

// One TICK is 2ms
#define TEN_MS                             5u
#define ONE_HUNDRED_MS                    50u
#define ONE_SECOND                       500u
#define TEN_SECONDS                     5000u

void     timer0_initialize (void);
unsigned current_tick (void);
unsigned timer0_elapsed_count (unsigned int start_tick);
void     timer0_wait (unsigned int num_tick);
void     ResetTickCount( void );
```

4.9 Source File: TIMER.C

```
DOS C51 COMPILER V5.10, COMPILATION OF MODULE TIMER
OBJECT MODULE PLACED IN TIMER.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE TIMER.C DB OE OR LARGE

stmt level       source

    1            // Timer.C
    2            //
    3            // P80C652 microcontroller general purpose timer routines
    4            //
    5            #include ”Timer.h”
    6            #include ”reg652.h”
    7
    8            /*-------------------------------------------------------------------
    9             Constant Declarations
   10             Every 2 ms, TIMER0 OVERFLOW and an interrupt will occur once
   11            ---------------------------------------------------------------------
   12            */
   13
   14            // 2ms tick: 10000h – ((11,059,200 Hz/12) * 0.002) = 0xF8CD
   15            #define TIMER0_HI   (unsigned char) 0xF8
   16            #define TIMER0_LO   (unsigned char) 0xCD
   17
   18            /*
   19            ---------------------------------------------------------------------
   20            Local Variable Declarations
   21            ---------------------------------------------------------------------
   22            */
   23
   24            static unsigned xdata timer0_tick;
   25
   26            /*
   28            ---------------------------------------------------------------------
   29            static void timer0_isr (void);
   30
   30            This function is an interrupt service routine for TIMER 0.  It should
   31            never be called by a C or assembly function.  It will be executed
   32            automatically when TIMER 0 overflows.
   34            ---------------------------------------------------------------------
   35            */
   37
```

4.9 Source File: TIMER.C (Continued)

```
        static void timer0_isr (void) interrupt 1 using 1
        {
 1
 1      /*------------------------------------------------
 1      Adjust the timer 0 counter so that we get another
 1      interrupt in 2 ms.
 1      ------------------------------------------------*/
 1       TR0 = 0;           /* stop timer 0 */
 1
 1      TL0 = TIMER0_LO;
 1      TH0 = TIMER0_HI;
 1
 1      TR0 = 1;            /* start timer 0 */
 1
 1      /*------------------------------------------------
 1      Increment the timer-tick.  This interrupt should
 1      occur approximately every 2ms.
 1      ------------------------------------------------*/
 1      timer0_tick++;
 1
 1      }

        /*
        ----------------------------------------------------------------------
        void timer0_initialize (void);

        set TIMER0 AS: mode 1; 16bit timer

        This function enables TIMER 0.  TIMER 0 will generate a synchronous
        Interrupt once every 2 ms.
        ----------------------------------------------------------------------
        */

        void timer0_initialize (void)
        {
 1      EA = 0;          /* disable interrupts */
 1
 1      timer0_tick = 0;
 1
 1      TR0 = 0;         /* stop timer 0 */
 1
 1      TMOD &= ~0x0F;   /* clear timer 0 mode bits */
 1      TMOD |= 0x01;    /* put timer 0 into 16–bit no prescale */
 1
 1      TL0 = TIMER0_LO;
 1      TH0 = TIMER0_HI;
 1
 1      PT0 = 0;  /* set low priority for timer 0, PT0 is in IP register */
 1      ET0 = 1;  /* enable timer 0 interrupt, ET0 is in IE register */
 1
 1      TR0 = 1;  /* start timer 0, TR0 is in TCON register */
 1
 1      EA = 1;   /* enable interrupts */
 1      }
```

4.9 Source File: TIMER.C (Continued)

```

             /*
             ---------------------------------------------------------------------
             void ResetTickCount( void );

             This function resets the timer-tick variable to zero.  The function
             should be used before each time the timer is used to time an event to
             prevent incorrect operation due to the timer0_tick variable rolling
             over to zero.  This will work out to a maximum of 64K * 2ms, or 128
             seconds.
             ----------------------------------------------------------------------
             */

             void ResetTickCount( void )
             {
    1        timer0_tick = 0;
    1        }

             /*
             ---------------------------------------------------------------------
             unsigned current_tick (void);

             This function returns the current timer0 tick count.
             ---------------------------------------------------------------------
             */

             unsigned current_tick (void)
             {
    1        unsigned xdata t;
    1
    1        EA = 0;
    1        t = timer0_tick;
    1        EA = 1;
    1
    1        return (t);
    1        }
```

4.9 Source File: TIMER.C (Continued)

```

           /*
           ---------------------------------------------------------------------
           unsigned timer0_elapsed_count (unsigned count);

           This function returns the number of timer-ticks that have elapsed since
           the specified count.
           ---------------------------------------------------------------------
           */

           unsigned timer0_elapsed_count( unsigned start_tick )
           {
1          return (current_tick( ) – start_tick);





1          }

           /*-------------------------------------------------------------------
           void timer0_wait ( unsigned count );

           This function waits for ’count’ timer-ticks to pass.
           ---------------------------------------------------------------------
           */
           void timer0_wait( unsigned num_tick )
           {
1             unsigned xdata test1;
1             unsigned xdata start_count;
1
1             start_count = current_tick ();
1
1             while( ( test1 = timer0_elapsed_count( start_count ) ) <= num_tick)
1             {
2             }
1          }


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

4.10 Source-Code Module: VMI5020

This module contains the TVP5020 VMI routines.

4.10.1 Vertical Blanking Interval Data Processor (VDP)

VDP registers are defined as all registers in the address range 90h – C2h. Non-VDP registers are defined as all other registers. There are different mechanisms for accessing VDP registers and non-VDP registers.

4.10.1.1 Accessing Non-VDP Registers

When writing to a non-VDP register by writing to the VMI data register, a long latency will occur before the TVP5020 is ready to write another data byte. This latency is at most one horizontal sync period (on the order of 64 µs). After this latency period, the operation complete interrupt is generated. This can be tested by polling the VMI status register, or by using the interrupt request signal (INTREQ) as is done in this program. After receiving the interrupt, the TVP5020 may initiate another VMI write cycle by writing to the VMI data register again.

Also, when loading the VMI address register with the address of a *readable* non-VDP register, a long latency will occur before the data byte is ready for the TVP5020 to read. This latency is at most one horizontal sync period. After this latency period, the operation complete interrupt is generated. This can be tested by polling the VMI status register, or by using the interrupt request signal (INTREQ) as is done in this program. After receiving the interrupt, the VMI data register may be read to obtain the read data.

4.10.1.2 Accessing VDP Registers

When writing to a VDP register by writing to the VMI data register, a long latency will not occur before the TVP5020 is ready for another data byte to be written and the operation complete interrupt will not be generated. After completion of the VMI write cycle, another VMI write cycle may be initiated immediately by writing to the VMI data register again.

Also, when loading the VMI address register with the address of a *readable* VDP register, a long latency will not occur before the data byte is ready for the TVP5020 to read, and the operation complete interrupt will not be generated. The VMI data register may be read immediately to obtain the read data.

4.10.2 Function: DecoderReset() (Lines 20–34)

This function configures the TVP5020 interrupt output (INTREQ) as required for accessing the non-VDP registers. First, register C1h is loaded with the value 20h. This enables the operation complete interrupt. Second, register C2h is loaded with the value 05h. This sets the INTREQ pin to active high (bit 0 = 1) and enables the YUV outputs (bit 2 = 1).

The YUV enable must also be set in the miscellaneous controls register (address 03h bit 4 = 1); otherwise the YUV outputs will be in the high-impedance state. This bit (address 03h bit 4) gets set when the TVP5020 registers get patched after the microcode download and CPU restart. Finally, FFh is written to the status register to reset all interrupt status bits and initialize the INTREQ pin to inactive (0).

4.10.3 Function: HandleDownload() (Lines 36–48)

This function is called to download the microcode to the TVP5020 program memory. Two parameters are passed to this function. Parameter *nCount* is the number of data bytes to write (plus 1 for the address byte). Parameter *pInBuf* is a pointer to the data to be written.

For the first VMI transaction, the first byte in the data buffer must be the address for microcode downloads (7Eh). *WriteTVP5020VMI()* is called to write the address to the VMI address register.

The global variable *g_bVDPAddrRange* is set to FALSE because the data will be written to a non-VDP register.

For the second VMI transaction, *WriteTVP5020VMI()* is called again to write the entire microcode data stream to the VMI data register. The third parameter is set to *FLAG_DATA_ONLY* to indicate that the data buffer starts with a data byte (not an address) and that the VMI address register does not need to be loaded.

4.10.4 Function: Write TVP5020() (Lines 50–56)

This function is called to write to a TVP5020 register. Two parameters are passed to this function. Parameter *nData* is the number of data bytes to write (plus 1 for the address byte). Parameter *pBuf* is a pointer to the data to be written. The *WriteTVP5020()* function does nothing but pass on parameters to the WriteTVP5020VMI() function. This enables the I2C5020, VMI5020 and VIP5020 source-code modules to be used interchangeably without having to change the rest of the program.

4.10.5 Function: Write TVP5020VMI() (lines 58–127)

This function performs a VMI write cycle for both VDP and non-VDP registers. Four parameters are passed in:

- *nLen* is the number of data bytes to write (plus 1 for the address byte—unless the *FLAG_DATA_ONLY* flag is set).
- *pBuf* is a pointer to the caller’s data buffer.
- *nFlags* takes on the values 0 or *FLAG_DATA_ONLY.*
  - 0 —First byte of data buffer is the address, load it into the VMI address register.
  - *FLAG_DATA_ONLY* —First byte of data buffer is a data byte, do not load the VMI address register.
- *nCtl* indicates the VMI register where data bytes will be written.

4.10.5.1 Address Phase (Lines 70–86)

If the *FLAG_DATA_ONLY* flag is set, the number of data bytes to write is the same as the *nLen* count passed. The VMI address register is not loaded.

Otherwise, subtract 1 from *nLen* to get the number of data bytes. Get the address from the first byte in the data buffer and point the data pointer to the first data byte. Determine the VDP/non-VDP status. Load the VMI address register.

4.10.5.2 Data Phase (Lines 88–125)

Jump to the proper code segment based on the VDP/non-VDP status.

For non-VDP Registers (Lines 90–117) perform the following steps:

1. Clear the interrupt status register and set INTREQ inactive.
2. Test and wait for INTREQ inactive.
3. Write a data byte from the caller's data buffer to the VMI data register.
4. Test and wait for INTREQ active.
5. Repeat steps 1 thru 4 for all data bytes.
6. Clear the interrupt status register and set INTREQ inactive.

For VDP Registers (Lines 119–125), perform the following steps:

1. Write a data byte from the caller's data buffer to the VMI data register.
2. Repeat step 1 for all data bytes.

4.10.6 Function: ReadSwitch() (Lines 129–154)

This function is TVP56000EVM-specific. It reads the logic levels of the DIP switches through the defined I/O pins of the P80C652. The result is packed into an 8-bit return value.

4.10.7 Function: ReadTVP5020() (Lines 156–168)

This function can be called to read from a TVP5020 register. The function is not used in this program, but is included here for reference. Three parameters are passed to this function. Parameter *nLength* is the number of data bytes to read. Parameter *pBuf* is a pointer to the caller's buffer where the read data is to be stored. Parameter *nSubAddr* is the address to read from.

The *ReadTVP5020()* function does nothing but pass on parameters to the *ReadTVP5020VMI()* function. This enables the I2C5020, VMI5020, and VIP5020 source-code modules to be used interchangeably without having to change the rest of the program.

4.10.8 Function: ReadTVP5020VMI() (Lines 170–220)

This function performs a VMI read cycle for both VDP and non-VDP registers. Three parameters are passed to this function:

1. *nLen* is the number of data bytes to read.
2. *pBuf* is a pointer to the caller's data buffer where read data is to be stored.
3. *nSubAddr* is the address to read from.

4.10.8.1 Address Phase

Determine the VDP/non-VDP status. Jump to the proper code segment based on the VDP/ non-VDP status.

For non-VDP Registers (Lines 186–190) perform the following steps:

1. Clear the interrupt status register and set INTREQ inactive.
2. Load the VMI address register.

For VDP Registers (Lines 212–213), load the VMI address register.

4.10.8.2 Data Phase

For non-VDP Registers (Lines 192–208) perform the following steps:

1. Test and wait for INTREQ active.
2. Clear the interrupt status register and set INTREQ inactive
3. Test and wait for INTREQ inactive.
4. Read a data byte from the VMI data register into caller’s data buffer.
5. Repeat steps 1 thru 4 for all data bytes.

For VDP Registers (Lines 214–218) perform the following steps:

1. Read a data byte from the VMI data register into caller’s data buffer.
2. Repeat step 1 for all data bytes.

4.11 Header File: VMI5020.H

```
// VMI5020.H
//
// Header file for VMI bus routines
//

#define FALSE                   0
#define TRUE                    !FALSE

#define ADDRESS   0
#define DATA      1
#define FIFO      2
#define STATUS    3

#define ACTIVE    1
#define INACTIVE  0

#define FLAG_DATA_ONLY 1

#define BOARD_TVP56000EVM   5

void          DecoderReset( void );
void          HandleDownload( unsigned nCount, unsigned char* pInBuf );
unsigned      WriteTVP5020(int nData, unsigned char *pBuf );
int           WriteTVP5020VMI( int nLen, unsigned char *pBuf, int nFlags, int nCtl );
unsigned char ReadSwitch( void );

#if(0)
unsigned    ReadTVP5020(int nLength, unsigned char *pBuf, unsigned char nSubAddr );
int         ReadTVP5020VMI( int nLen, unsigned char *pBuf, int nSubAddr );
#endif
```

4.12 Source File: VMI5020.C

```
DOS C51 COMPILER V5.10, COMPILATION OF MODULE VMI5020
OBJECT MODULE PLACED IN VMI5020.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE VMI5020.C DB OE OR LARGE

stmt level    source

   1          //
   2          // VMI5020.C
   3          //
   4          // Routines for TVP5020 using VMI Bus
   5          //
   6
   7          #include ”VMI5020.h”
   8          #include ”Reg652.h”
   9
  10
  11          // Identify the board
  12          unsigned char      g_nBoardID = BOARD_TVP56000EVM;
  13
  14          // Memory–Mapped TVP5020 VMI Ports
  15          extern unsigned char xdata g_pVMI[];
  16
  17          // If TRUE, INTREQ handshake is not required
  18          unsigned char g_bVDPAddrRange = FALSE;
  19
  20          void DecoderReset( void )
  21          {
  22   1          // Initialize interrupts
  23   1          // Enable operation complete interrupt
  24   1          g_pVMI[ADDRESS] = 0xC1;
  25   1          g_pVMI[DATA]    = 0x20;
  26   1
  27   1          // INTREQ is active high, enable YUV outputs
  28   1          g_pVMI[ADDRESS] = 0xC2;
  29   1          g_pVMI[DATA]    = 0x05;
  30   1
  31   1          // Reset interrupts
  32   1          g_pVMI[STATUS]  = 0xFF;
  33   1          return;
  34   1      }
  35
  36          void HandleDownload( unsigned nCount, unsigned char* pInBuf )
  37          {
  38   1          // Write the sub–address byte
  39   1          WriteTVP5020VMI( 1, pInBuf++, 0, DATA );
  40   1
  41   1          // Write the data as one block – data bytes only
  42   1          g_bVDPAddrRange = FALSE;
  43   1
  44   1          // FLAG_DATA_ONLY =  Buffer contains data bytes only
  45   1          WriteTVP5020VMI( nCount–1, pInBuf, FLAG_DATA_ONLY, DATA );
  46   1
  47   1          return;
  48   1      }
  49
```

4.12 Source File: VMI5020.C (Continued)

```
        unsigned WriteTVP5020(int nData, unsigned char *pBuf )
        {
1           // Write data via 8–bit memory mapped port
1           WriteTVP5020VMI( nData, pBuf, 0, DATA );
1
1           return( nData );

1       }

        int WriteTVP5020VMI( int nLen, unsigned char *pBuf, int nFlags, int
                              nCtl )
        {
1           // nLen     – byte count including sub–address byte and data
            //            bytes OR just data bytes
1           // pBuf     – pointer to caller’s buffer
1           // nFlags   – FLAG_DATA_ONLY
1           // nCtl     – The VMI register to write data to
1
1           int j = 0;
1           int nAddr = 0;
1           int nDataLen = 0;
1           unsigned char* pInData = pBuf;
1
1           // Determine number of data bytes
1           if( nFlags & FLAG_DATA_ONLY )
1           {
2               nDataLen = nLen;
2           }
1           else
1           {
2               nDataLen = nLen – 1;
2               // Point to first data byte
2               pInData++;
2               // Get address from buffer
2               nAddr = *pBuf;
2               // Save the VDP / non–VDP status
2               g_bVDPAddrRange = ( ( nAddr >= 0x90 ) && ( nAddr <= 0xC2 ) )?
                                  TRUE : FALSE;
2               // Write the address
2               g_pVMI[ADDRESS] = nAddr;
2           }
1
```

4.12 Source File: VMI5020.C (Continued)

```
1          // Write ’nDataLen’ bytes
1          if( !g_bVDPAddrRange )
1          {
2              for(j=0; j<nDataLen; j++ )
2              {
3                   // When there is no address to write there is no way to
3                   // know ’bVDPAddrRange’.  Caller sets that variable
                    // before calling WriteTVP5020VMI()
3
3                   // Reset interrupts
3                   g_pVMI[STATUS] = (unsigned char)0xFF;
3                   while( INTREQ != INACTIVE )
3                   {
4                       ;
4                   }
3                   // Write the data
3                   g_pVMI[nCtl] = *pInData++;
3
3                   // If not a VDP register, wait for the interrupt that
3                   // signals that the write operation has completed
3                   while( INTREQ != ACTIVE )
3                  {
4                       ;
4                  }
3              }
2
2              // Reset interrupts
2              g_pVMI[STATUS] = (unsigned char)0xFF;
2          }

1          else
1          {
2              for(j=0; j<nDataLen; j++ )
2              {
3                  // Write the data
3                  g_pVMI[nCtl] = *pInData++;
3              }
2          }
1          return( 0 );
1      }
```

4.12 Source File: VMI5020.C (Continued)

```
         unsigned char ReadSwitch( void )
         {
   1           unsigned char nVal = 0;
   1
   1           // See REG652.H for SW1...SW3 definitions
   1           SW1 = 1;     //MSB
   1           SW2 = 1;
   1           SW3 = 1;     //LSB
   1
   1           if ( SW1 == 1 )
   1           {
   2              nVal += 4;
   2           }
   1
   1           if ( SW2 == 1 )
   1           {
   2              nVal += 2;
   2           }
   1
   1           if ( SW3 == 1 )
   1           {
   2              nVal += 1;
   2           }
   1
   1           return( nVal );
   1     }

```

4.12 Source File: VMI5020.C (Continued)

```
#if(0)
// Note: This is how to read a TVP5020 register via VMI
unsigned ReadTVP5020(int nLength, unsigned char *pBuf,
                    unsigned char nSubAddr )
{
   // nLength   = # data bytes to read
   // pBuf      = Pointer to where first data word read should be
   //             stored
   // nSubAddr = sub–address to read

   ReadTVP5020VMI( nLength, pBuf, nSubAddr );
   return( 0u );
}

int ReadTVP5020VMI( int nLen, unsigned char *pBuf, int nSubAddr )
{
   // nLen     – number of bytes to read
   // pBuf     – pointer to data buffer where read data is to be
   //            stored
   // nSubAddr – sub–address to read from

   int j = 0;

   // Save the VDP / non–VDP status
   g_bVDPAddrRange = ((nSubAddr >= 0x90) && (nSubAddr <= 0xC2)) ?
                     TRUE : FALSE;

   // Write the address
   if( !g_bVDPAddrRange )
   {
          // Reset interrupts
          g_pVMI[STATUS] = (unsigned char)0xFF;

          // Write the read address – should set INTREQ
          g_pVMI[ADDRESS] = nSubAddr;

          for(j=0; j<nLen; j++ )
          {
             while( INTREQ != ACTIVE )
             {
                    ;
             }

             // Reset interrupts
             g_pVMI[STATUS] = (unsigned char)0xFF;
             while( INTREQ != INACTIVE )
             {
                    ;
             }

             // Read the data – should set INTREQ
             *pBuf++ = g_pVMI[nCtl];
          }
   }
```

4.12 Source File: VMI5020.C (Continued)

```
        else
        {
            // Write the read address – should not set INTREQ
            g_pVMI[ADDRESS] = nSubAddr;
            for(j=0; j<nLen; j++ )
            {
                // Read the data – should not set INTREQ
                *pBuf++ = g_pVMI[nCtl];
            }
        }
        return( 0 );
    }
    #endif


C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

4.13 Source-Code Module: I2C6000

This module contains the TVP6000-specific I2C routines and initialization data for the TVP6000.

4.13.1 Function: read_tvp6000() (lines 23–40)

This function can be called to read from a TVP6000 register. The function is not used in this program, but is included here for reference. Three parameters are passed to this function. Parameter *read_length* is the number of data bytes to read. Parameter *sub_addrs* is the subaddress to read from. Parameter *output_buf* is a pointer to the caller’s buffer where the read data is to be stored.

For the TVP6000, there is no restriction on the number of bytes read per I2C transaction. The I2C function *start_i2c()* is called first, with the TVP6000 I2C device ID for writes, to write the subaddress value. Then, *start_i2c()* is called again, with the TVP6000 I2C device ID for reads, to read the requested number of data bytes.

4.13.2 Function: write_tvp6000() (Lines 41–50)

This function is called to write to a TVP6000 register. Two parameters are passed to this function. Parameter *write_length* is the number of data bytes to write (plus 1 for the subaddress byte). Parameter *input_buf* is a pointer to the data to be written. The first byte in the data must be the subaddress. For the TVP6000, there is no restriction on the number of bytes written per I2C transaction. The I2C function *start_i2c()* is called to perform the I2C data transfer. The TVP6000 I2C device ID for writes is passed as a parameter.

4.13.3 Function: LoadTVP6000() (Lines 52–75)

This function initializes the TVP6000 registers. The parameter *nMode* indicates the video mode that will be set up. The register data for the selected video mode is written to the TVP6000 by passing the array name of the proper register data set to *write_tvp6000()*. Finally, the function PatchTVP6000() is called.

4.13.4 Function: PatchTVP6000 (Lines 77–97)

This function performs an initialization step that is necessary to properly initialize the YUV interface. The YUV format is momentarily changed to 16-bit YUV 4:2:2 mode and then back to the 8-bit ITU-R BT.656 interface mode.

The global variable *g_nBoard* is tested to determine the value to write into subaddress 3Ah: 0Fh to use video port 1, or 1Fh to use video port 2. This code was inserted only to make the same code work on an EVM prototype board that uses VP1 instead of VP2. Normally, this *if* statement should be taken out.

4.14 Header File: I2C6000.H

```
// I2C6000.H
//
// Header file for TVP6000 routines
//
#define ENCODER_WRITE    0x42 // I2C writing address of encoder
#define ENCODER_READ     0x43 // I2C reading address of encoder
#define NUM_OF_REGISTER  0x65 // No of TVP6000 registers to load
#define BOARD_TVP56000EVM   5
#define BOARD_EVM3          3

#if(0)
void read_tvp6000(int length_of_read, unsigned char first_subaddress,
            unsigned char* readout_buffer);
#endif

void write_tvp6000(int length_of_write, unsigned char *write_buf);
void LoadTVP6000( int nMode );
void PatchTVP6000( void );
```

4.15 Source File: I2C6000.C

```
DOS C51 COMPILER V5.10, COMPILATION OF MODULE I2C6000
OBJECT MODULE PLACED IN I2C6000.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE I2C6000.C DB OE OR LARGE

stmt level    source

  1          //
  2          // I2C6000.C
  3          //
  4          // Routines for TVP6000 using I2C Bus
  5          //
  6          #include ”I2C.h”
  7          #include ”I2C6000.h”
  9          #include ”timer.h”
  8          #include ”DATA6000.h”
 10
 11          extern unsigned char      g_nBoardID;
 12
 13          static unsigned char patch1_6000[2] =
 14          {
 15             0x3A, 0x0D    //16–bit YUV 4:2:2, color bar TPG OFF
 16          };
 17
 18          static unsigned char patch2_6000[2] =
 19          {
 20             0x3A, 0x0F    // CCIR 656, VP1
 21          };
 22
 23          #if(0)
             void read_tvp6000(int read_length, unsigned char sub_addrs,
                               unsigned char *output_buf)
             {
                 unsigned char *sub_begin;
                 sub_begin = &sub_addrs;

                 initia_i2c();   /* initialize I2C bus */

                 /* this will write the first SUB–address to TVP6000 */
                 /* after this, data will be read out start from this address */
                 start_i2c(ENCODER_WRITE, 1, sub_begin);

                 /* read all registers of TVP6000, and put them into a temporary
                    buffer */
                 start_i2c(ENCODER_READ, read_length, output_buf);
                 return;
             }
             #endif
 40
```

4.15 Source File: I2C6000.C (Continued)

```
       void write_tvp6000(int write_length, unsigned char *input_buf)
       {
1          initia_i2c();   /* initialize I2C bus */
1
1          // This will write the first sub–address to TVP6000
1          // after this, data will be read out starting from this address
1          // subaddress will be 1st element in the input_buf
1          start_i2c(ENCODER_WRITE, write_length, input_buf);
1          return;
1      }

       void LoadTVP6000( int nMode )
       {
1          switch( nMode )
1          {
2              case 0:
2                  write_tvp6000(NUM_OF_REGISTER, T600N601);
2              break;
2
2              case 1:
2                  write_tvp6000(NUM_OF_REGISTER, T600NSQP);
2              break;
2
2              case 2:
2                  write_tvp6000(NUM_OF_REGISTER, T600P601);
2              break;
2
2              case 3:
2                  write_tvp6000(NUM_OF_REGISTER, T600PSQP);
2              break;
2          }
1          PatchTVP6000();
1          return;
1      }

       void PatchTVP6000( void )
       {
1         unsigned nStartPoint = 0;
1
1          write_tvp6000( 2, patch1_6000 );
1
1          // Delay for 100ms
1         nStartPoint = current_tick ();
1         while( timer0_elapsed_count( nStartPoint ) < ONE_HUNDRED_MS )
1         { ; }
1         if( g_nBoardID == BOARD_TVP56000EVM )
1         {
2           // Use video port 2 for TVP56000EVM
2           patch2_6000[1] = 0x1F;
2         }
1          write_tvp6000( 2, patch2_6000 );
1         return;
1      }

C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

4.16 TVP6000 Initialization Data for NTSC with CCIR601 Sampling

```
// DATA6000.H
// Header file containing all TVP6000 register initialization data.
//
unsigned char code T600N601[] = {
      /* Register Name                         Sub–Address */
0x3A, /* SUB–ADDRESS                                N/A    */
0x0F, /* F_CONTROL                                  3A     */
      /* RESERVED                                          */
                  0x00, 0x00, 0x00, 0x00, 0x00,  /* 3B–3F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 40–47  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 48–4F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 50–57  */
0x00, 0x00,                                      /* 58–59  */
0x00, /* C_PHASE                                    5A     */
0x0E, /* GAIN_U                                     5B     */
0x7D, /* GAIN_V                                     5C     */
0xCE, /* BLACK_LEVEL                                5D     */
0xB8, /* BLANK_LEVEL                                5E     */
0x31, /* GAIN_Y                                     5F     */
0x20, /* X_COLOR                                    60     */
0x0D, /* M_CONTROL                                  61     */
0x3A, /* BSTAMP                                     62     */
0x1F, /* S_CARR1                                    63     */
0x7C, /* S_CARR2                                    64     */
0xF0, /* S_CARR3                                    65     */
0x21, /* S_CARR4                                    66     */
0x00, /* LINE21_O0                                  67     */
0x00, /* LINE21_O1                                  68     */
0x00, /* LINE21_E0                                  69     */
0x00, /* LINE21_E1                                  6A     */
0x12, /* LN_SEL                                     6B     */
0x00, /* SYN_CTRL0                                  6C     */
0x40, /* RCML21                                     6D     */
0xF2, /* HTRIGGER0                                  6E     */
0x00, /* HTRIGGER1                                  6F     */
0xC0, /* VTRIGGER                                   70     */
0x89, /* BMRQ                                       71     */
0x39, /* EMRQ                                       72     */
0x61, /* BEMRQ                                      73     */
0x08, /* X2PH                                       74     */
0x90, /* X1PH                                       75     */
0x00, /* RESERVED                                   76     */
0xEA, /* BRCV                                       77     */
0x8A, /* ERCV                                       78     */
0x60, /* BERCV                                      79     */
0x0C, /* FLEN                                       7A     */
0x06, /* FAL                                        7B     */
0x06, /* LAL                                        7C     */
0x22, /* FLAL                                       7D     */
0x0E, /* SYN_CTRL1                                  7E     */
      /* RESERVED                         0x00      7F     */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 80–87  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 88–8F  */
      /* Scaling Processor Registers                       */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 90–97  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00               /* 98–9D  */  };
```

4.17 TVP6000 Initialization Data for NTSC with Square Pixel Sampling

```
unsigned char code T600NSQP[] = {
      /* Register Name                         Sub–Address */
0x3A, /* SUB–ADDRESS                                N/A    */
0x0F, /* F_CONTROL                                  3A     */
      /* RESERVED                                          */
                  0x00, 0x00, 0x00, 0x00, 0x00,  /* 3B–3F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 40–47  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 48–4F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 50–57  */
0x00, 0x00,                                      /* 58–59  */
0x00, /* C_PHASE                                    5A     */
0x0E, /* GAIN_U                                     5B     */
0x7D, /* GAIN_V                                     5C     */
0xCE, /* BLACK_LEVEL                                5D     */
0xB8, /* BLANK_LEVEL                                5E     */
0x31, /* GAIN_Y                                     5F     */
0x20, /* X_COLOR                                    60     */
0x0D, /* M_CONTROL                                  61     */
0xBA, /* BSTAMP                                     62     */
0x55, /* S_CARR1                                    63     */
0x55, /* S_CARR2                                    64     */
0x55, /* S_CARR3                                    65     */
0x25, /* S_CARR4                                    66     */
0xA5, /* LINE21_O0                                  67     */
0x50, /* LINE21_O1                                  68     */
0xA5, /* LINE21_E0                                  69     */
0x50, /* LINE21_E1                                  6A     */
0x14, /* LN_SEL                                     6B     */
0x24, /* SYN_CTRL0                                  6C     */
0x40, /* RCML21                                     6D     */
0xDE, /* HTRIGGER0                                  6E     */
0x00, /* HTRIGGER1                                  6F     */
0xDF, /* VTRIGGER                                   70     */
0xBF, /* BMRQ                                       71     */
0xBF, /* EMRQ                                       72     */
0x50, /* BEMRQ                                      73     */
0x08, /* X2PH                                       74     */
0x90, /* X1PH                                       75     */
0x00, /* RESERVED                                   76     */
0xD3, /* BRCV                                       77     */
0xD3, /* ERCV                                       78     */
0x50, /* BERCV                                      79     */
0x0C, /* FLEN                                       7A     */
0x06, /* FAL                                        7B     */
0x06, /* LAL                                        7C     */
0x22, /* FLAL                                       7D     */
0x0E, /* SYN_CTRL1                                  7E     */
      /* RESERVED                         0x00      7F     */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 80–87  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 88–8F  */
      /* Scaling Processor Registers                       */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 90–97  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00               /* 98–9D  */  };
```

4.18 TVP6000 Initialization Data for PAL with CCIR601 Sampling

```
unsigned char code T600P601[] = {
      /* Register Name                         Sub–Address */
0x3A, /* SUB–ADDRESS                                N/A    */
0x0F, /* F_CONTROL                                  3A     */
      /* RESERVED                                          */
                  0x00, 0x00, 0x00, 0x00, 0x00,  /* 3B–3F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 40–47  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 48–4F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 50–57  */
0x00, 0x00,                                      /* 58–59  */
0x00, /* C_PHASE                                    5A     */
0x15, /* GAIN_U                                     5B     */
0x8C, /* GAIN_V                                     5C     */
0xBC, /* BLACK_LEVEL                                5D     */
0xBC, /* BLANK_LEVEL                                5E     */
0x45, /* GAIN_Y                                     5F     */
0x20, /* X_COLOR                                    60     */
0x0E, /* M_CONTROL                                  61     */
0x41, /* BSTAMP                                     62     */
0xCB, /* S_CARR1                                    63     */
0x8A, /* S_CARR2                                    64     */
0x09, /* S_CARR3                                    65     */
0x2A, /* S_CARR4                                    66     */
0xA2, /* LINE21_O0                                  67     */
0x2A, /* LINE21_O1                                  68     */
0xA2, /* LINE21_E0                                  69     */
0x2A, /* LINE21_E1                                  6A     */
0x14, /* LN_SEL                                     6B     */
0x00, /* SYN_CTRL0                                  6C     */
0x00, /* RCML21                                     6D     */
0x16, /* HTRIGGER0                                  6E     */
0x01, /* HTRIGGER1                                  6F     */
0x80, /* VTRIGGER                                   70     */
0x5F, /* BMRQ                                       71     */
0x5F, /* EMRQ                                       72     */
0x61, /* BEMRQ                                      73     */
0x08, /* X2PH                                       74     */
0x90, /* X1PH                                       75     */
0x00, /* RESERVED                                   76     */
0x0E, /* BRCV                                       77     */
0xAE, /* ERCV                                       78     */
0x61, /* BERCV                                      79     */
0x70, /* FLEN                                       7A     */
0x05, /* FAL                                        7B     */
0x35, /* LAL                                        7C     */
0x22, /* FLAL                                       7D     */
0x0E, /* SYN_CTRL1                                  7E     */
      /* RESERVED                         0x00      7F     */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 80–87  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 88–8F  */
      /* Scaling Processor Registers                       */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 90–97  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00               /* 98–9D  */  };
```

4.19 TVP6000 Initialization Data for PAL with Square Pixel Sampling

```
unsigned char code T600PSQP[] = {
      /* Register Name                         Sub–Address */
0x3A, /* SUB–ADDRESS                                N/A    */
0x0F, /* F_CONTROL                                  3A     */
      /* RESERVED                                          */
                  0x00, 0x00, 0x00, 0x00, 0x00,  /* 3B–3F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 40–47  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 48–4F  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 50–57  */
0x00, 0x00,                                      /* 58–59  */
0x00, /* C_PHASE                                    5A     */
0x15, /* GAIN_U                                     5B     */
0x8C, /* GAIN_V                                     5C     */
0xBC, /* BLACK_LEVEL                                5D     */
0xBC, /* BLANK_LEVEL                                5E     */
0x45, /* GAIN_Y                                     5F     */
0x20, /* X_COLOR                                    60     */
0x0E, /* M_CONTROL                                  61     */
0xC1, /* BSTAMP                                     62     */
0x0C, /* S_CARR1                                    63     */
0x8C, /* S_CARR2                                    64     */
0x79, /* S_CARR3                                    65     */
0x26, /* S_CARR4                                    66     */
0xA2, /* LINE21_O0                                  67     */
0x2A, /* LINE21_O1                                  68     */
0xA2, /* LINE21_E0                                  69     */
0x2A, /* LINE21_E1                                  6A     */
0x14, /* LN_SEL                                     6B     */
0x00, /* SYN_CTRL0                                  6C     */
0x00, /* RCML21                                     6D     */
0x32, /* HTRIGGER0                                  6E     */
0x01, /* HTRIGGER1                                  6F     */
0x80, /* VTRIGGER                                   70     */
0x5F, /* BMRQ                                       71     */
0x5F, /* EMRQ                                       72     */
0x61, /* BEMRQ                                      73     */
0x08, /* X2PH                                       74     */
0x90, /* X1PH                                       75     */
0x00, /* RESERVED                                   76     */
0x28, /* BRCV                                       77     */
0x28, /* ERCV                                       78     */
0x71, /* BERCV                                      79     */
0x70, /* FLEN                                       7A     */
0x05, /* FAL                                        7B     */
0x35, /* LAL                                        7C     */
0x22, /* FLAL                                       7D     */
0x0E, /* SYN_CTRL1                                  7E     */
      /* RESERVED                         0x00      7F     */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 80–87  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 88–8F  */
      /* Scaling Processor Registers                       */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  /* 90–97  */
0x00, 0x00, 0x00, 0x00, 0x00, 0x00               /* 98–9D  */  };
```