Design Stellaris 2006 Contest
Project Number LM1633 by Lindsay Meek
Luminary Micro Component LM3S811
Complete Documentation

Uninterruptible Solar Power Supply

Introduction

The gradual warming of our planet due to man-made pollution is a topical issue. There are record high levels of greenhouse gas emissions, and 20% of these gas emissions can be attributed to residential energy use[1]. One component of residential energy is the power used by standby loads, such as answering machines and cordless phones. The plug pack used to power such devices runs 24 hours a day, 7 days a week, consuming a small but constant amount of energy. Over a year, this figure can add up to a surprisingly significant amount of energy, for example 27 kWh for the typical answering machine[2]. This equates to roughly 27 metric kilograms (43 lbs) of CO_2 gas generated from a coal-fired power station. The US EPA estimated in 1999 there were 200 million answering machine/cordless phones installed, which equated to a staggering 5½ billion metric kilograms of greenhouse gas produced per year.

The uninterruptible solar power supply is a device intended to take these small standby loads off the grid, and power them directly from solar panels. The device acts as an uninterruptible power supply (UPS), switching the load back to the grid if the solar power is unavailable during the night or cloudy weather.

One advantage of using a solar-powered UPS is that it bypasses the usual regulatory requirements imposed by utilities on grid-interactive solar power systems, as the solar panels are never directly connected to the grid and pose no possible risk to its stability. Another advantage of focusing on powering only standby loads is the system cost is kept low, requiring in most installations only a single solar panel.

Interfaces

The Solar UPS design is shown in the diagram below. The solar energy is captured by a solar panel, and supplied to the inverter as a DC voltage. The inverter then converts the DC to an AC voltage, where it can be used to supply the standby load.

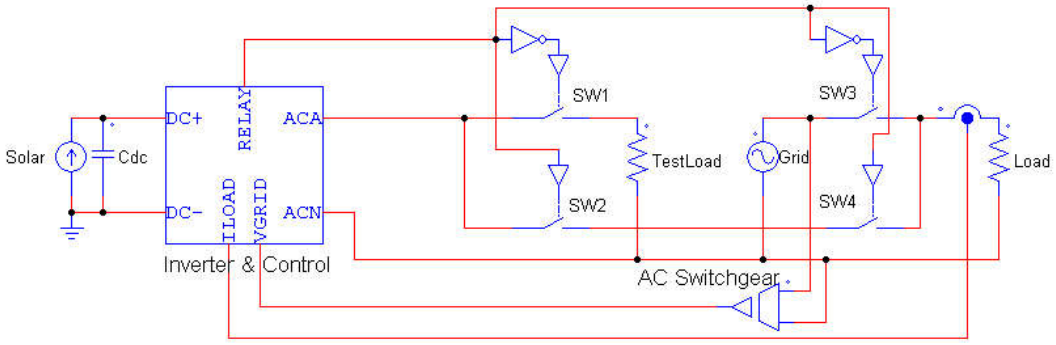
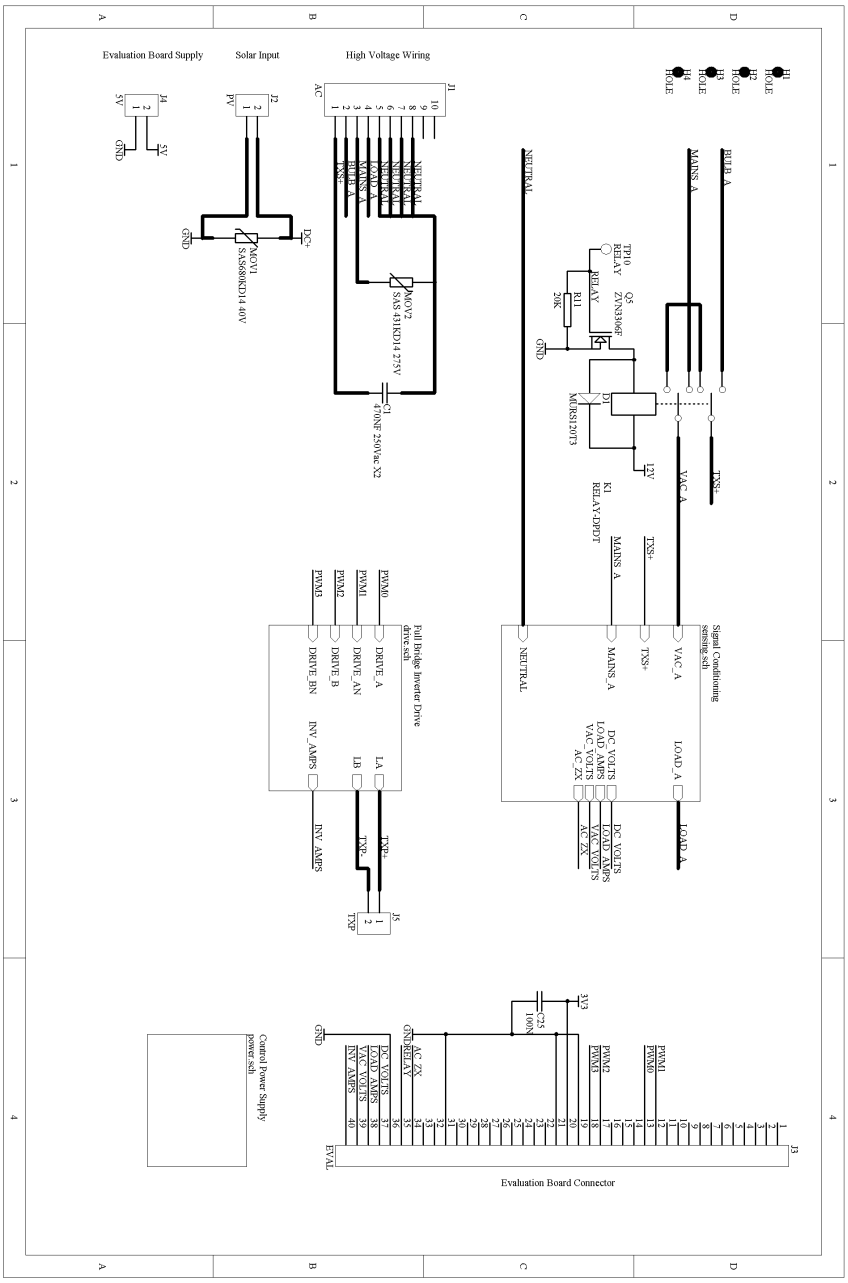


Figure 1. System Diagram

The SPDT relay contact indicated in the diagram by SW3 and SW4 provides the UPS function, by switching the standby load between the conventional grid and the solar inverter output. The solar output is selected only if the solar is sufficient to supply the standby load.

The second SPDT relay contact indicated in the diagram by SW1 and SW2 redirects the inverter output to a test load when the inverter is not connected to the standby load. This test load is used during a test sequence to determine the amount of available solar power. The test sequence varies the inverter output voltage to determine how much power the solar panel can supply before its DC voltage drops off. The maximum available solar power can then be compared to the standby load to determine if the load demand can be met by the solar. If this is the case, and the inverter output is synchronised to the grid voltage, then the UPS changes the standby load over to solar power by closing the relay.

Top Level Schematic



The top level schematic for the solar UPS shows the specific interconnections indicated on the system diagram.

Connections

A series of screw terminals are used to attach to the various AC and DC flying leads. The solar comes in on connector J2 where it is then converted into low voltage AC by the inverter

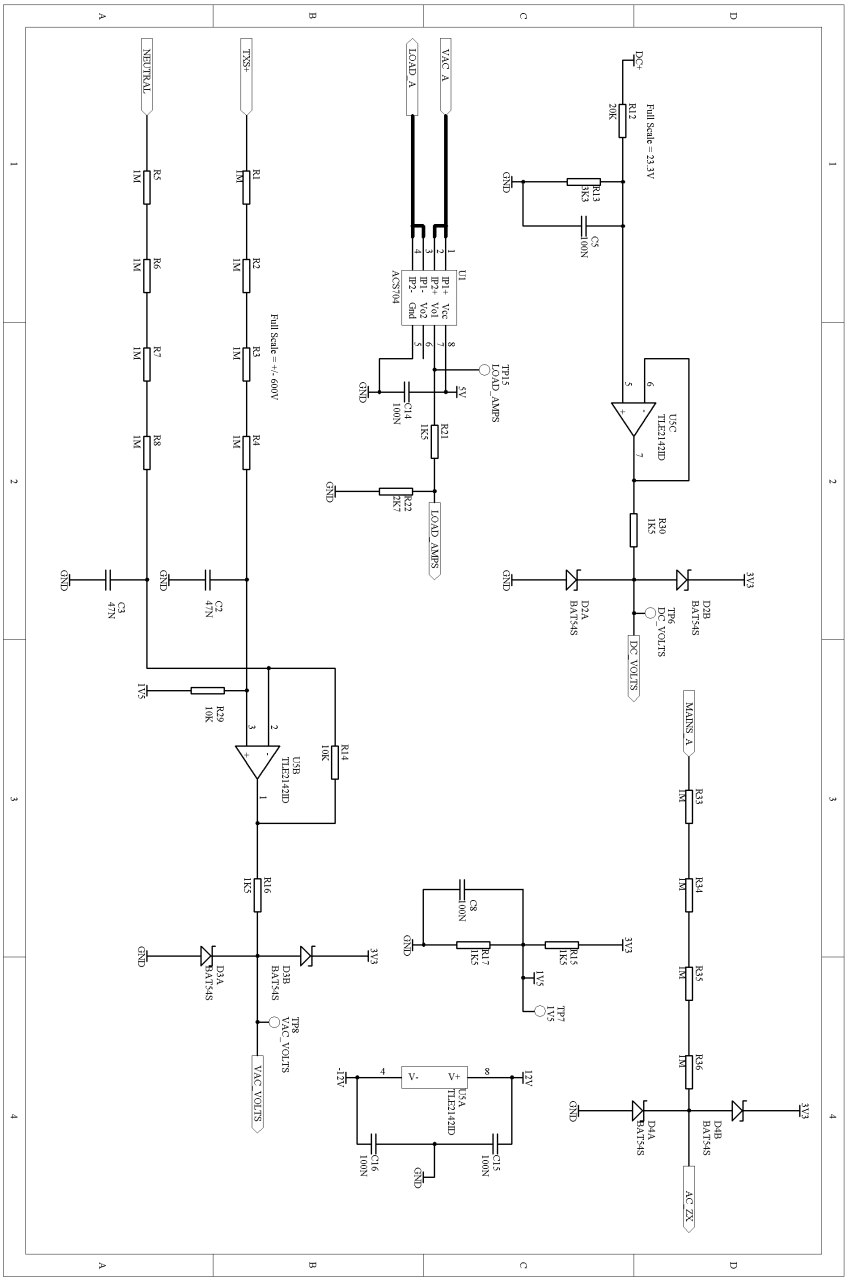
and passed out to the transformer primary via connector J5. The high voltage AC from the secondary of the transformer comes back on connector pins J1/1 and J1/5 where it is filtered by capacitor C1 and switched through the UPS changeover relay K1.

The test load is normally connected to the transformer output via connector pins J1/2 and J1/6 under control of K1.

The grid input comes in on connector pins J1/3 and J1/7, where it again switched through K1 to the standby load connected on J1/4 and J1/8.

The other system connectors are used to attach the LM3S811 evaluation board, and these consist of single-in-line 0.1” pitch headers lining up with the evaluation board connectors.

Signal Conditioning



The various analog system measurements require level conversion from their input voltage ranges to the 0...3.3V range acceptable to the microcontroller's analog-to-digital converter (ADC). Additionally, frequencies above 20kHz are attenuated to reduce aliasing noise during sampling.

DC Voltage Sensing

The DC voltage is scaled using a resistive divider so that 3.3Vdc at the ADC corresponds to 23.3Vdc at the input. This is also filtered by an RC filter with its cutoff frequency set to ~600Hz, isolated by a voltage-following op-amp and clamped by some protection diodes at the output to prevent overloading of the ADC during transients.

AC Voltage Sensing

The inverter AC output voltage output is stepped down using four 1MEG resistors and scaled so that ± 600V corresponds to ± 1.5V, and a DC offset of 1.5V is also added to eliminate negative voltages going into the ADC. Four resistors in series are used instead of one as the resistors break down with more than 150V across them.

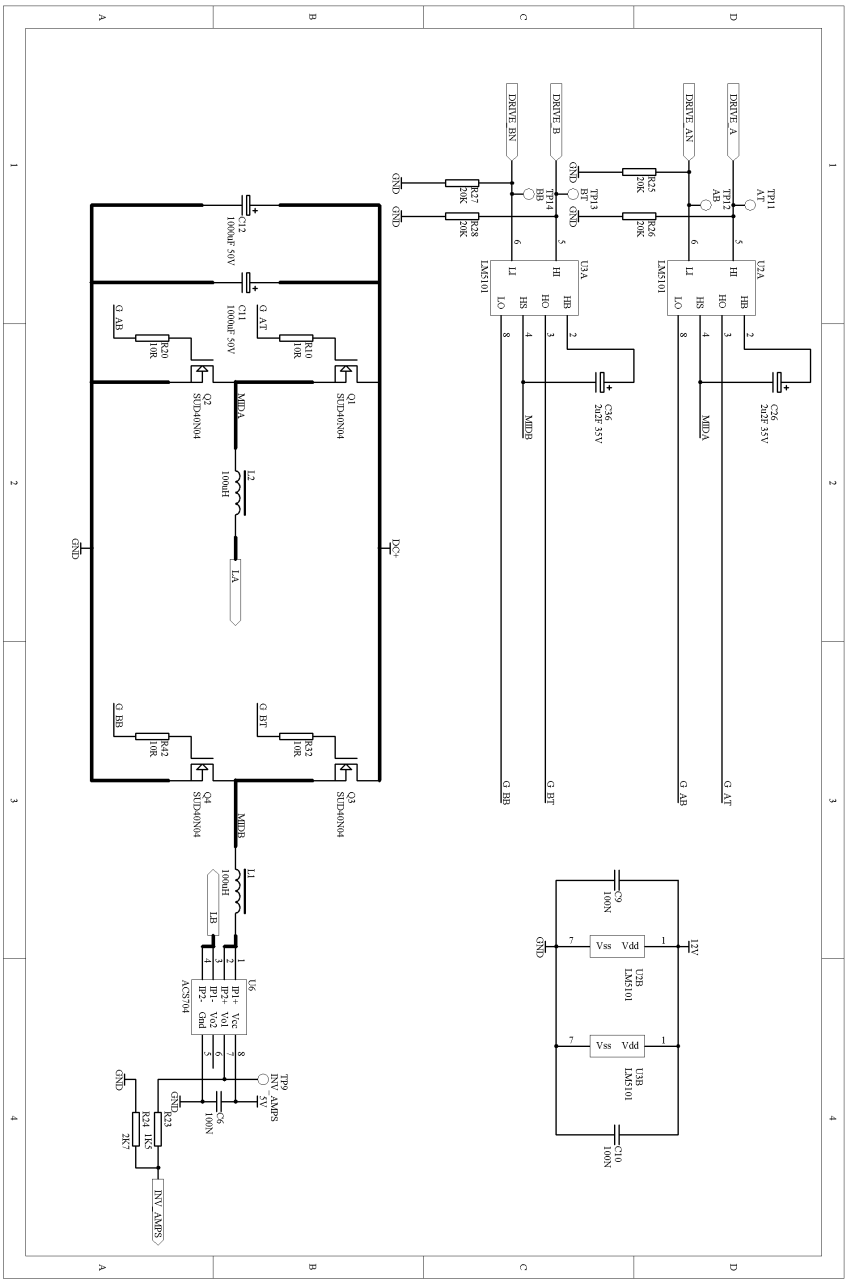
Load & Inverter Current Sensing

Two hall-effect current sensors are used to measure the current for the load and inverter, and these provide convenient 0-5V outputs, which are scaled down to 3.3V using resistive dividers and connected directly to the ADC inputs.

Grid Frequency & Phase Measurement

The grid frequency and phase are measured by monitoring a digital representation of the grid voltage zero crossing, using two voltage clamping diodes applied to the grid voltage and connected to the supply rails. The current through these diodes is limited by four 1MEG resistors connected in series, in a similar arrangement to the AC voltage sensing resistors.

Inverter Power Stage



The inverter power stage is responsible for converting the DC voltage from the solar panels to low-voltage AC, which is then stepped up to the grid voltage using an external toroidal transformer.

Topology

A conventional full-bridge buck topology is used to perform the inversion. This topology sinusoidally modulates either the left or right half switches of the bridge using a 20kHz pulse width modulation (PWM). This generates a positive and negative alternating potential at the transformer, creating the desired AC waveform.

Components

Four low on-resistance MOSFET switches are used in the bridge to maximise its efficiency, and these MOSFETs are controlled using two half-bridge gate drive ICs. The resulting output waveform has the 20 kHz switching component removed using an LC filter.

A hall-effect current sensor is placed in the output circuit to allow the control to measure the inverter current. This measurement is used in conjunction with a test load to determine the amount of available solar energy.

Efficiency

The inverter efficiency can be estimated by summing the losses in the power circuit. The dominant losses are due by the Control SMPS, the MOSFETs, the HF inductor DC resistance and the transformer losses. These are predicted at full load of 50W to be

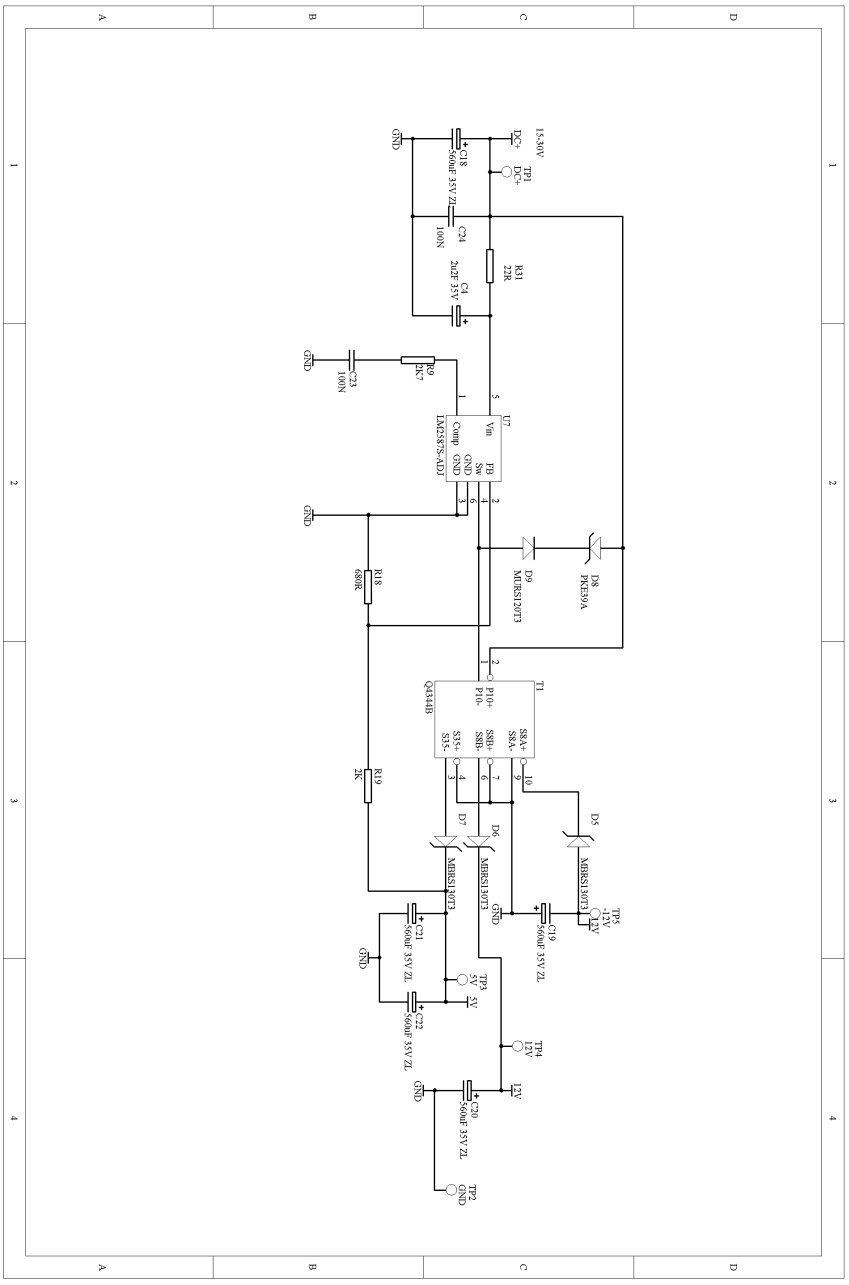
MOSFETs	~0.3W
Inductors	~0.7W
Transformer	<10 W
SMPS	~3.3W
Predicted Total losses	<14.3W
Predicted Efficiency	>71%

The measured losses were slightly better than the predicted values:

Measured Total losses	~10.3W @ 40W output
Measured Efficiency	~78% @ 40W

Minor improvements in the efficiency can be made in a future design revision by removing some of the unnecessary devices on the evaluation board, thereby reducing power consumption by the control SMPS. The transformer can also be customised for higher efficiency operation however this may add to the cost of the unit.

Control Power Supply



A small, switched mode power supply is used to generate the regulated voltage rails for the control circuits. The control power requirements for the solar UPS are quite small, specifically

Rail	**Worst Case Current**
12V	15mA
-12V	5mA
5V	520mA

Or, a total power of approximately 3 watts.

Topology

The main challenge of the power supply is to ensure it can operate with an input voltage below the level of the rails. For this reason, a flyback topology was chosen.

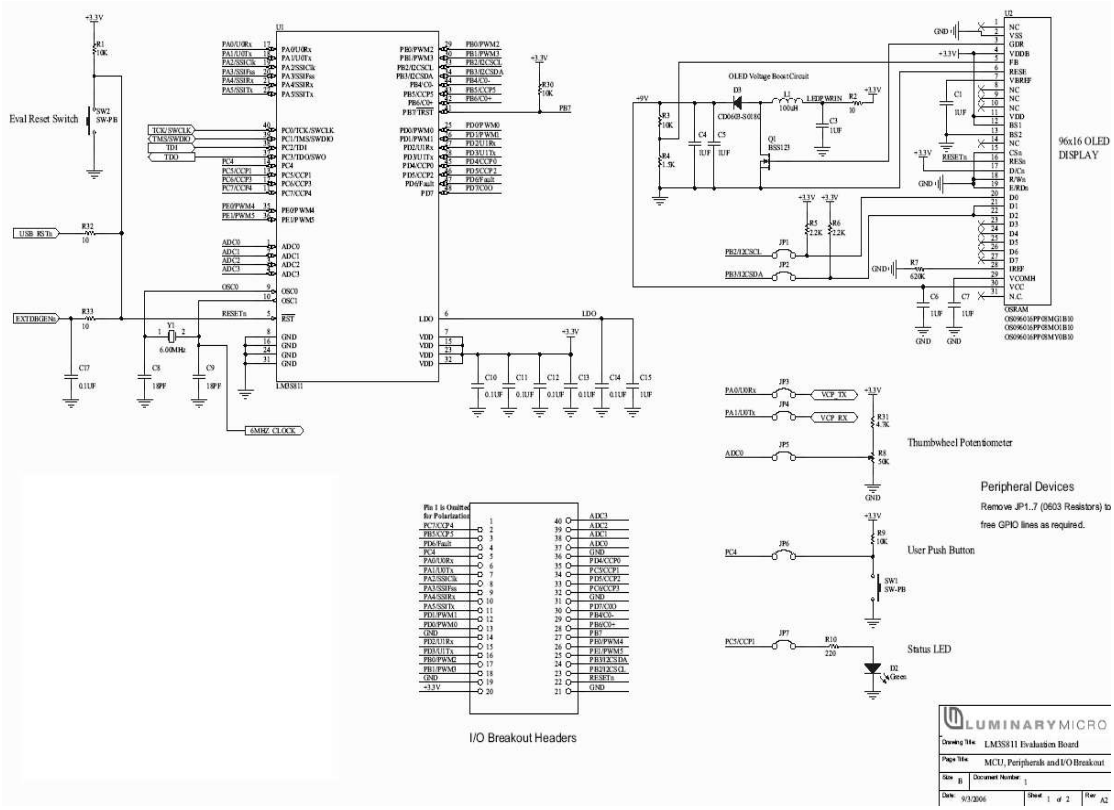
Components

The National Semiconductor's LM2587S was selected as this was relatively simple to design with and off-the-shelf matched transformers are available.

Limitations

It was found during testing that the duty cycle range of the off-the-shelf transformer created a limitation on the minimum input voltage, causing DC voltages below 14V to draw excessive switching currents. This situation can be avoided in practise somewhat by using thin-film solar panels such as Kaneka's P-LE55[3]. These panels maintain their output voltage better when they reach operating temperature. In a subsequent revision, it is recommended that a flyback transformer with a higher turns ratio be used.

Microcontroller



The LM3S811 evaluation board is used as the microcontroller for the solar UPS. This connected using a daughter board configuration where two rows of pins are used to attach the evaluation board to the main board. The 5Vdc power supply was fed via a butchered USB cable from the control SMPS on the main board.

Some clearances proved very tight near the base of the evaluation board, requiring some more butchering of the PCB to make it fit. In a revised version, the LM3S811 microcontroller should be fitted directly to the main board and powered from a 3.3Vdc supply, saving considerable space.

Analog to Digital Conversion

The LM3S811 has an onboard 10-bit 4-channel analog to digital converter (ADC), which is used to sample and measure the analog channels of the UPS; Input DC voltage, Inverter AC output voltage, Inverter AC primary current, and Load AC current. These channels are scaled using signal conditioning circuits to the 0-3V range required by the ADC.

The ADC on the LM3S811 is configured to sample all four channels automatically and store the results in a FIFO, which helps to reduce CPU loading. The triggering of the ADC is also configured to occur automatically at the mid-point of the PWM generator.

After the ADC completes converting the four analog channels, it triggers an interrupt allowing the CPU to gather the results and process them.

For measuring the AC levels, an approximate RMS algorithm is used which sums the absolute sample values rather than the multiplied sample values. This speeds up the processing by eliminating the need to multiply accumulate every sample, and it can be shown using calculus that the RMS summation is 1.11 times the absolute summation.

Note that the ADC is operated in unipolar mode, so the DC offsets of the AC channels needs to measured and subtracted prior to calculating the absolute values.

Inverter Waveform Generation

The inverter waveform consists of a unipolar switching scheme, with one 'leg' or side of the inverter bridge held low for 180 degrees of the AC waveform (the standing phase). The other leg of the bridge is modulated with a sinusoidal PWM, with a modulation depth proportional to the desired output voltage. This process reverses every 180 degrees to generate an alternating voltage.

The LM3S811 PWM0 & 1 blocks are used to directly generate the gate drive signals for all four switches of the full bridge, as it incorporates dead-time generation for preventing cross-conduction in the switches. The PWM is configured to generate an up-down PWM as this has lower harmonics than an aligned PWM.

The ADC interrupt is reused to update the duty cycle of the PWM blocks, as this interrupt is synchronised PWM and using a single interrupt service routine reduces CPU loading.

An index into a 512-entry sinusoidal lookup table is multiplied by a gain term to calculate the instantaneous duty cycle. The index into the lookup table is incremented by an differential step proportional to the desired output AC frequency and the interrupt frequency:

Index Step = Output AC Frequency * 512 / Interrupt Frequency

Phase Locked Loop

The inverter AC waveform must be in phase with the grid AC waveform, otherwise the transformer could be out of phase when the changeover relay is closed, causing a short circuit. To lock the inverter to grid, a simple zero-crossing based phase locked loop is used.

The phase locked loop measures the period of the grid waveform using the event capture feature on the LM3S811. The event capture stores the value of Timer1B and generates an interrupt whenever a negative to positive zero crossing is detected on the grid voltage input. The interrupt then subtracts the timer value from the last timer value to yield the grid period. Using this approach allows the frequency to be measured very accurately, however it can be susceptible to false zero crossing events caused by line noise. To reduce the effects of noise, excessively small or large period measurements are rejected.

A second timer is used to trap overflows on Timer1B, which can occur when the grid period exceeding 65535 timer ticks. In this case Timer1A operates in unison with Timer1B, and generates an underflow interrupt which increments a 16-bit register. This register is combined with the captured timer value when a zero crossing event occurs to give an equivalent 32-bit timer value.

The measured grid period is used to adjust the stepping rate of the inverter waveform generator until the inverter is generating the same period. The phase relationship is maintained by resetting the inverter waveform generator to zero degrees whenever a valid grid zero crossing occurs.

Phase lock is detected by measuring the phase angle of the inverter waveform generator when the grid zero crossing occurs. If the difference in phase angle is found to be sufficiently small (~5 degrees), the waveforms are considered locked.

Inverter Voltage Control

The inverter control loop consists of a closed loop voltage regulator, which measures the RMS output voltage of the inverter and regulates it to the desired output voltage by adjusting the gain of the inverter AC waveform. A closed loop control is necessary as the output of the inverter can vary depending on the load, due to internal voltage drops. The loop consists of a

simple integral regulator that runs every line cycle, comparing the measured output voltage to the desired output voltage, and adjusting the duty cycle.

Supervisory Control

In addition to controlling the inverter and locking it to the grid, a supervisory control is used which is responsible for managing the changeover relay and determining when the solar can supply the load.

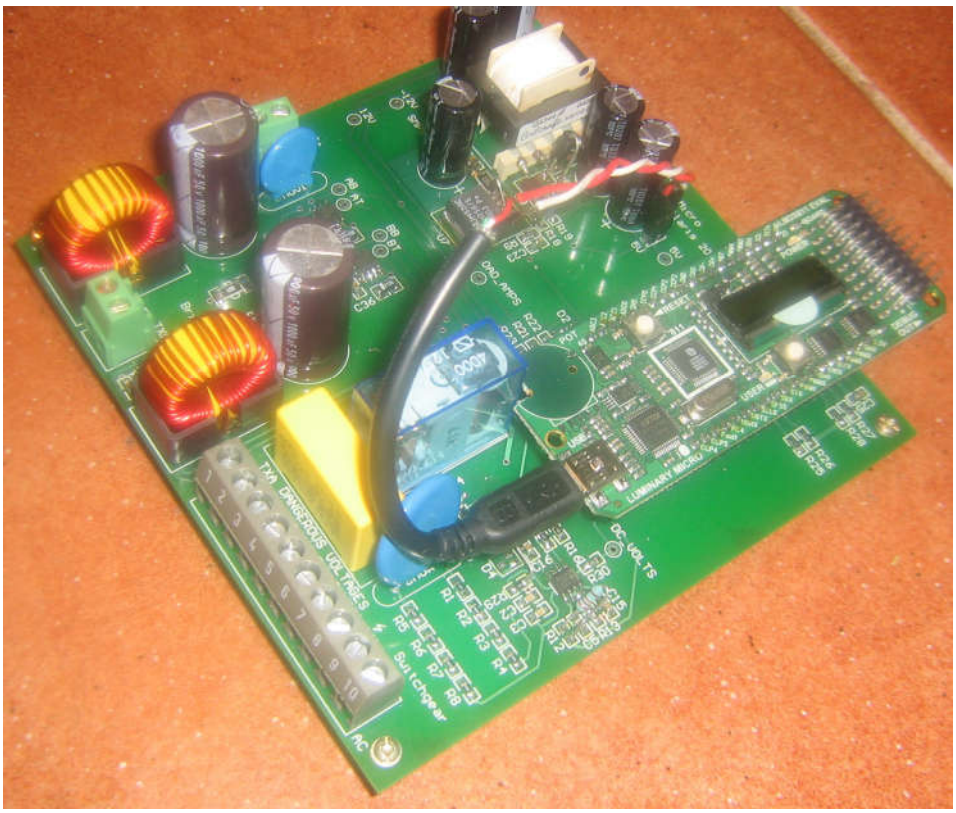
The available solar power is determined by connecting the inverter output to a test load, and ramping up the inverter voltage until it reaches the rated level of 230Vac. If the DC voltage drops below a minimum threshold during the ramping up period, the supervisory attempts to maintain this voltage to prevent the solar input voltage collapsing. This condition indicates that the inverter is unable to supply the test load. If the solar power increases, the ramping commences again and continues until the inverter output reaches the rated level or the DC voltage drops again.

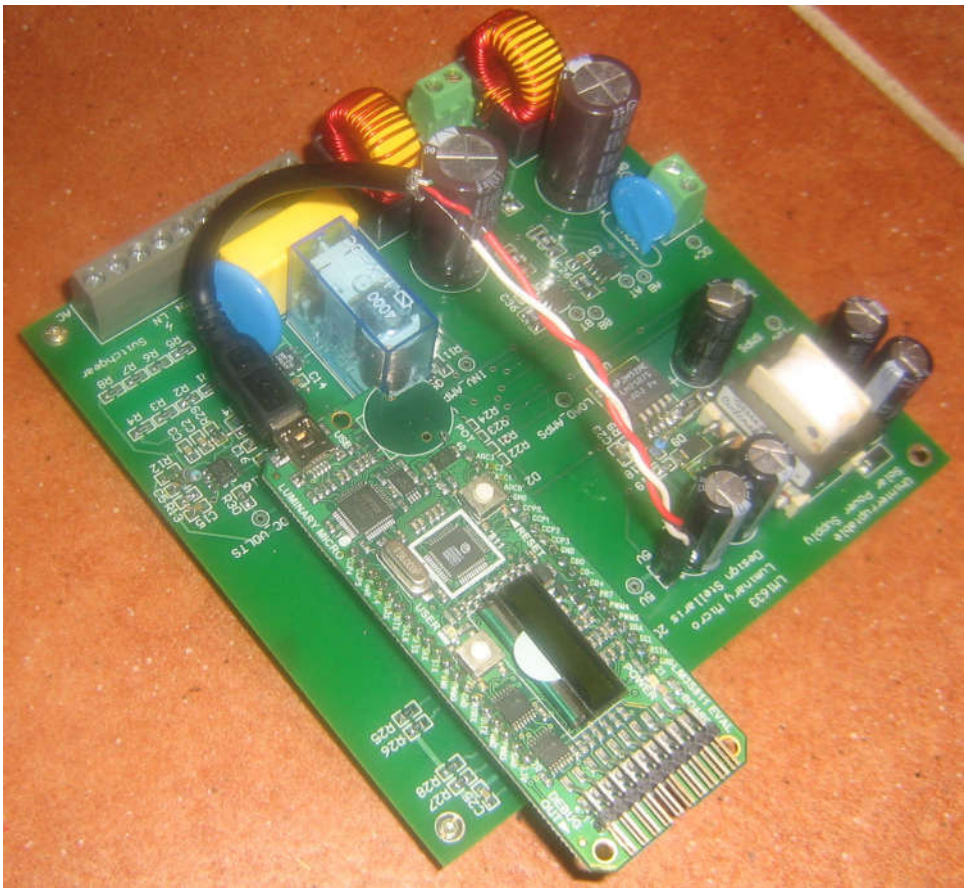
The supervisory control will not connect the solar to the standby load until the inverter has reached its rated voltage, and is locked to the grid voltage waveform. In addition, the load current is compared to the inverter primary current reflected to the secondary side of the transformer (divided by the transformer turns ratio). If the load current does not exceed the inverter current, then the changeover relay is closed.

Whilst the solar is connected to the standby load, the supervisory monitors the DC voltage against a threshold level and immediately disconnects the solar if the DC voltage drops suddenly. This can indicate that the standby load has suddenly increased, or the available solar power has decreased due to a lack of sunlight.

During testing, it was found that putting a large electrolytic capacitor on the DC bus helped stabilise the UPS during transients.

Project Photos





Conclusions & Further Work

There is something timeless about powering electronic devices directly from sunlight. The Solar UPS enables this on a small scale, by connecting household appliances to solar and avoiding the expense of batteries and the problem of sunlight not always being available.

The prototype detailed in this article performed its function as a small solar UPS with a reasonable efficiency, however some improvements could be made.

The low-frequency transformer topology used for the inverter had quite a low efficiency at the power level at which it was operating. This was due primarily to the toroidal transformer losses, and the somewhat by the current consumption of the evaluation board. A better design could eliminate the transformer to improve efficiency, however this approach may have difficulty with standards compliance due to the lack of isolation between the DC and AC voltages.

On a lesser note, power losses could be reduced by integrating the evaluation board electronics onto the main PCB, and eliminating some of the unnecessary devices.

The other problem encountered during testing was the limitation of panel voltage range on the control SMPS. A better design could employ a flyback transformer with a higher turns ratio and have a provision for under voltage shutdown.

References

[1] "Emissions of Greenhouse Gases in the United States in 2005". Energy Information Administration. http://www.eia.doe.gov/oiaf/1605/ggrpt/carbon.html

[2] "Proposal from the U.S. Environmental Protection Agency to the North American Communications Environmental Excellence Initiative and the Center for Resource Management". Environmental Protection Agency.
http://www.energystar.gov/ia/partners/prod_development/archives/downloads/phones/proposaltoindustry.pdf

[3] http://www.pv.kaneka.co.jp/products/index.html

Source Code Listing

LM1633.C

```
//*****************************************************************************
//      Design Stellaris Contest Entry LM1633
//      Uninterruptable Solar Power Supply
//*****************************************************************************

// includes
#include "hw_ints.h"
#include "hw_memmap.h"
#include "hw_types.h"
#include "hw_pwm.h"
#include "debug.h"
#include "gpio.h"
#include "pwm.h"
#include "sysctl.h"
#include "diag.h"
#include "osram96x16.h"
#include "watchdog.h"
#include "adc.h"
#include "timer.h"
#include "interrupt.h"
#include <stdio.h>
#include <math.h>
#include <string.h>

// types
typedef unsigned int Word32;
typedef signed int SWord32;
typedef unsigned short Word16;
typedef signed short SWord16;
typedef unsigned char BYTE;
typedef BYTE bool;

// defines

// program flag bits
#define FLAG_BUTTON           0       // Debounced state of the button
#define FLAG_DEBOUNCE_LOW     1       // Low bit of the debounce clock
#define FLAG_DEBOUNCE_HIGH    2       // High bit of the debounce clock
#define FLAG_BUTTON_PRESS     3       // The button was just pressed
#define FLAG_SWITCHING        4       // Switching is enabled
#define FLAG_LOCK             5       // Locked to mains waveform
#define FLAG_ZXPHASE          6       // Phase polarity of mains
#define FLAG_RELAY            7       // Mirror of relay state
#define FLAG_NCOPHASE         8       // Phase polarity of NCO
#define FLAG_RATED            9       // At rated voltage flag
#define FLAG_LOAD_HIGH        10      // Load too high flag

//*****************************************************************************
//
// The GPIOs for the push button, user LED and relay control.
//
//*****************************************************************************
#define PUSH_BUTTON           GPIO_PIN_4
#define USER_LED              GPIO_PIN_5
#define RELAY                 GPIO_PIN_4
#define MAINS_ZX              GPIO_PIN_6

// Transformer low side voltage RMS
#define TXPRI_VOLTAGE 9
// Transformer high side voltage RMS
#define TXSEC_VOLTAGE 240

// Desired output voltage RMS (before changover)
#define OUTPUT_VOLTAGE 230

// Square root of 2 constant
```

```
#define SQRT2 1.41421356237

// Multiplier to convert sum of absolute value to approximate RMS value (PI/(2*Sqrt2))
#define ABS2RMS 1.1107207345

// Full scale DC voltage
#define VDC_FULLSCALE 23.3

// Minimum DC voltage
#define VDC_MIN 16.0

// Full scale DC current
#define IDC_FULLSCALE 4.34

// Full scale AC voltage
#define VAC_FULLSCALE 600.0

// Analog channel allocations
#define CHAN_VDC 0
#define CHAN_LOAD_AMPS 1
#define CHAN_VAC 2
#define CHAN_INV_AMPS 3
// Pseudo channel
#define CHAN_FREQ 4
#define CHAN_MAINS 5

// Minimum duty cycle
#define MINIMUM_DUTY 20

//
// main control interrupt rate
//
#define TIMER_FREQ 20000

/// Nominal AC frequency in Hz
#define NOM_FREQ 50

// Minimum AC lock frequency in Hz
#define MIN_FREQ 45

// Maximum AC lock frequency in Hz
#define MAX_FREQ 65

// Frequency of capture timer in Hz
#define CAPTURE_FREQ 20000000Lu

// Minimum AC lock period in jiffies
#define MIN_PERIOD (CAPTURE_FREQ / MAX_FREQ)
// Maximum AC lock period in jiffies
#define MAX_PERIOD (CAPTURE_FREQ / MIN_FREQ)

// nominal phase velocity as 8.16 fixed point value
#define NOM_STEP (((Word32)NOM_FREQ << 25)/TIMER_FREQ)

// The phase lock range 512 = 360 degrees
// 5 degrees
#define PHASE_LOCK_RANGE 7

/// one second worth of ADC measurements to determine DC offsets
#define INTEGRATIONS 20000

// variables


//*****************************************************************************
//
// A set of flags used to track the state of the application.
//
//*****************************************************************************
static volatile Word32 g_ulFlags;     // Program flag bits
static SWord32 Period;                // 100% PWM duty value
static volatile Word32 index=0;       // NCO phase << 16
static volatile Word32 index_step;    // NCO phase velocity
static Word16 indexb;                 // NCO sample index (0..511)
static SWord16 vac_gain;              // Gain of voltage component
static volatile Word32 ticker;        // Timer ticker
static SWord32 MAXIMUM_DUTY;          // Maximum duty cycle
```

```
static SWord16 dc_offsets[4];        // DC offsets of AC channels

// Mark position for last zero crossing
static Word32 zx_mark=0;

// Period of ZX in timer0 ticks
static volatile Word32 zx_period;

// lock decision integrator
static Word16 pll_cnt=128;

// flag indicating mains ZX event to background task
static char mains_zx=0;

// Current output voltage
static SWord16 OutputVoltage=0;

// Current output voltage scaled to raw units
static SWord16 OutputVoltageRaw=0;

// Incoming ADC sample buffer and then some
static unsigned long ulADC[16];

// sum of absolute value ADC accumulators(approx rms)
static Word32 sum_abs[4];

// number of ADC samples taken
static Word16 sum_cnt;

// latched average absolute ADC values
static Word16 sum_abs_latched[4];

// 512-entry sine wave lookup normalised to 15 bits
extern const SWord16 sinlut[];

//*****************************************************************************
// The error routine that is called if the driver library encounters an error.
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

// Control the AC relay open (0) or closed (1)
static void SetRelay(BYTE level)
{
        GPIODirModeSet(GPIO_PORTD_BASE, RELAY, GPIO_DIR_MODE_OUT);

        HWREGBITW(&g_ulFlags, FLAG_RELAY)=level;

        if(level)
                GPIOPinWrite(GPIO_PORTD_BASE, RELAY, RELAY);
        else
                GPIOPinWrite(GPIO_PORTD_BASE, RELAY, 0);
}

// Stop switching
static void StopSwitching(void)
{
        HWREGBITW(&g_ulFlags, FLAG_SWITCHING)=0;
}

// Start switching
static void StartSwitching(void)
{
        vac_gain = 0;
        HWREGBITW(&g_ulFlags, FLAG_SWITCHING)=1;
}

// Set the output voltage of the inverter stage (Volts RMS)
static void SetOutputVoltage(SWord16 voltage)
{
        SWord32 t;

        OutputVoltage = voltage;
```

```
        // compute raw level for direct comparison to vac

        t = voltage;

        t <<= 17;

        t = t / (SWord32)(ABS2RMS * 256.0 * VAC_FULLSCALE);

        OutputVoltageRaw = (SWord16) t;
}

// Main UPS control algorithm .. runs every zero crossing
static void ControlLoop(void)
{
        SWord16 t;

        // is the relay closed (PV supplying load?)
        if(HWREGBITW(&g_ulFlags, FLAG_RELAY))
        {
                // Below minimum voltage?
                if(ulADC[CHAN_VDC] < (Word16)(VDC_MIN * 1024.0 / VDC_FULLSCALE))
                {
                        // Yes, connect back to mains
                        SetRelay(0);
        // Set output to zero immediately to prevent reset due to supply under voltage
                        SetOutputVoltage(0);

                        vac_gain=0;
                }
        }
        else
        {
                HWREGBITW(&g_ulFlags, FLAG_RATED)=0;

                // try to reach request voltage without lowering Vdc below minimum
                if(ulADC[CHAN_VDC] > (Word16)(VDC_MIN * 1024.0 / VDC_FULLSCALE))
                {
                        // DC is ok, try increasing AC
                        if(OutputVoltage < OUTPUT_VOLTAGE)
                                ++OutputVoltage;
                        else
                        {
                                // Set flag indicating we at rated voltage
                                HWREGBITW(&g_ulFlags, FLAG_RATED)=1;
                        }

                }
                else
                {
                        // DC is in trouble, lower output voltage
                        if(OutputVoltage > 0)
                                --OutputVoltage;

                }

                // Load new operating point
                SetOutputVoltage(OutputVoltage);

                // is dump load current > output load current transformed
                // to primary side
                if(sum_abs_latched[CHAN_INV_AMPS]/(TXSEC_VOLTAGE/TXPRI_VOLTAGE) >=
                                sum_abs_latched[CHAN_LOAD_AMPS])
                        HWREGBITW(&g_ulFlags, FLAG_LOAD_HIGH)=0;
                else
                        HWREGBITW(&g_ulFlags, FLAG_LOAD_HIGH)=1;

                // locked to grid?
                if(HWREGBITW(&g_ulFlags, FLAG_LOCK))
                {
                        // yes, is voltage at rated?
                        if(HWREGBITW(&g_ulFlags, FLAG_RATED))
                        {
                                // yes, is dummy current > load current
                                if(HWREGBITW(&g_ulFlags, FLAG_LOAD_HIGH)==0)
                                {
                                        // yes, close relay and start supply load
```

```
                                        SetRelay(1);

                                }
                        }
                }
        }

        // closed loop output voltage regulator

        // control error
        t=OutputVoltageRaw - (SWord16)sum_abs_latched[CHAN_VAC];

        // control output
        vac_gain += t;

        // clip gain to allowable limits
        if(vac_gain    < 0)
                vac_gain=0;

        if(vac_gain > MAXIMUM_DUTY-20)
                vac_gain=MAXIMUM_DUTY-20;
}

// Long counter for ZX timer
static volatile Word16 timer16=0;

// Overflow interrupt for measuring periods exceeding 65535 timer ticks
void Timer1AIntHandler(void) __irq
{
    // Clear the timer interrupt.
    TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

        // bump zx timer most significant word
        timer16--;
}

// Interrupt for intercepting grid zero crossing captures
void Timer1BIntHandler(void) __irq
{
        static char toggle=0;
        Word32 temp;

        // Clear the timer interrupt.
        TimerIntClear(TIMER1_BASE, TIMER_CAPB_EVENT);

        // Compute time point from overflow counter and capture time
        temp = timer16;
        temp <<=16;
        temp |=TimerValueGet(TIMER1_BASE, TIMER_B) & 0xFFFFu;

        // Compute period since last capture expressed in 50ns ticks
        zx_period = zx_mark - temp;

        // Update capture point
        zx_mark = temp;

        // Indicate that a ZX event has occurred
        mains_zx = 1;

        // only allow acceptable frequency range
        if(zx_period < MAX_PERIOD && zx_period > MIN_PERIOD)
        {
                // Toggle the user LED as a diagnostic
                toggle ^= 1;

                if(toggle)
                        GPIOPinWrite(GPIO_PORTC_BASE, USER_LED, USER_LED);
                else
                        GPIOPinWrite(GPIO_PORTC_BASE, USER_LED, 0);

                         // work out ideal NCO velocity based on line period
                         temp =  0xFA000000Lu / (zx_period >> 3);

                         // check for lock condition
                         if(indexb > 512-PHASE_LOCK_RANGE || indexb < PHASE_LOCK_RANGE)
                         {
                                if(pll_cnt < 255)
```

```
                                        pll_cnt++;

                                if(pll_cnt > 192)
                                        HWREGBITW(&g_ulFlags, FLAG_LOCK)=1;
                        }
                        else
                        {
                                if(pll_cnt > 0)
                                        pll_cnt--;

                                if(pll_cnt < 64)
                                {
                                        if(HWREGBITW(&g_ulFlags, FLAG_LOCK))
                                        {
                                         // Force frequency back to nominal if
                                         // coming out of lock
                                         HWREGBITW(&g_ulFlags, FLAG_LOCK)=0;

                                         index_step=NOM_STEP;
                                        }
                                }
                        }

                        // lock NCO to line period (1st order control)
                        index_step = (index_step + temp)>>1;

                        // lock NCO phase (zero order control)
                        index=0;

        }

}

// The interrupt handler for the ADC interrupt.     This runs at 20kHz
void ADCIntHandler(void)  __irq
{
        SWord16 duty;
        SWord32 temp;
        char zx_flag;

        // Clear the ADC interrupt.
        ADCIntClear(ADC_BASE, 0);

        // Read the data from the ADC FIFO
        ADCSequenceDataGet(ADC_BASE, 0, ulADC);

        // Integrate the AC channels into the respective accumulators
        temp = ulADC[CHAN_LOAD_AMPS] - dc_offsets[CHAN_LOAD_AMPS];
        if(temp < 0) temp=-temp;
        sum_abs[CHAN_LOAD_AMPS] += temp;
        temp = ulADC[CHAN_INV_AMPS] - dc_offsets[CHAN_INV_AMPS];
        if(temp < 0) temp=-temp;
        sum_abs[CHAN_INV_AMPS] += temp;
        temp = ulADC[CHAN_VAC] - dc_offsets[CHAN_VAC];
        if(temp < 0) temp=-temp;
        sum_abs[CHAN_VAC] += temp;
        sum_cnt++;

        // If we are switching
        if(HWREGBITW(&g_ulFlags, FLAG_SWITCHING))
        {

        // advance NCO phase
        index += index_step;

        // extract sample index
        indexb = (index >> 16) & 511;

        // determine output duty cycle
        temp = ((SWord32)sinlut[indexb] * vac_gain) >> 15;
        duty = (SWord16)temp;

        // Determine standing phase
        if(indexb & 256)
        {
                // clip to minimum duty to stop PWM hardware glitch
                // from deadtime generator
```

```
        if(duty > -MINIMUM_DUTY)
                duty=-MINIMUM_DUTY;

        // Standing phase transition at 180 degrees?
        if(HWREGBITW(&g_ulFlags, FLAG_NCOPHASE))
        {
                HWREGBITW(&g_ulFlags, FLAG_NCOPHASE)=0;

                // Force bottom switch on on inactive bridge leg
                HWREG(PWM_BASE + PWM_GEN_1_OFFSET + PWM_O_X_GENA ) =
                        ((PWM_GEN_ACT_ONE << PWM_GEN_ACT_A_UP_SHIFT) |
                        (PWM_GEN_ACT_ZERO <<   PWM_GEN_ACT_A_DN_SHIFT)|
                        (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_ZERO_SHIFT) |
                        (PWM_GEN_ACT_ONE << PWM_GEN_ACT_LOAD_SHIFT));

                HWREG(PWM_BASE + PWM_GEN_0_OFFSET + PWM_O_X_GENA ) =
                        ((PWM_GEN_ACT_ZERO << PWM_GEN_ACT_A_UP_SHIFT) |
                        (PWM_GEN_ACT_ZERO <<   PWM_GEN_ACT_A_DN_SHIFT)|
                        (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_ZERO_SHIFT) |
                        (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_LOAD_SHIFT));

        }

        // Load duty cycle into active leg
        HWREG(PWM_BASE+PWM_GEN_1_OFFSET+PWM_O_X_CMPA ) = (Period + duty ) >> 1;

}
else
{
        // clip to minimum duty to stop PWM hardware glitch
        // from deadtime generator
        if(duty < MINIMUM_DUTY)
                duty=MINIMUM_DUTY;

        // Standing phase transition at zero degrees?
        if(!HWREGBITW(&g_ulFlags, FLAG_NCOPHASE))
        {
                HWREGBITW(&g_ulFlags, FLAG_NCOPHASE)=1;
                // Inform control that zero crossing has occurred
                zx_flag=1;

                // Force bottom switch on inactive bridge leg
                HWREG(PWM_BASE + PWM_GEN_0_OFFSET + PWM_O_X_GENA ) =
                        ((PWM_GEN_ACT_ONE << PWM_GEN_ACT_A_UP_SHIFT) |
                        (PWM_GEN_ACT_ZERO <<   PWM_GEN_ACT_A_DN_SHIFT) |
                        (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_ZERO_SHIFT) |
                        (PWM_GEN_ACT_ONE << PWM_GEN_ACT_LOAD_SHIFT));

                HWREG(PWM_BASE + PWM_GEN_1_OFFSET + PWM_O_X_GENA ) =
                        ((PWM_GEN_ACT_ZERO << PWM_GEN_ACT_A_UP_SHIFT) |
                        (PWM_GEN_ACT_ZERO <<   PWM_GEN_ACT_A_DN_SHIFT)|
                        (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_ZERO_SHIFT) |
                        (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_LOAD_SHIFT));
        }
        else
                zx_flag=0;

        // Load duty cycle into active bridge leg
        HWREG(PWM_BASE+PWM_GEN_0_OFFSET+PWM_O_X_CMPA ) = (Period - duty ) >> 1;
        // Zero crossing event?
        if(zx_flag)
        {
         // Yes, average sample measurements over the last line cycle
         // and reset accumulators
         if(sum_cnt)
         {
                temp = sum_abs[CHAN_LOAD_AMPS] / sum_cnt;
                if(temp >= 4) temp-=4;
                sum_abs_latched[CHAN_LOAD_AMPS] = (Word16)temp;
                temp = sum_abs[CHAN_INV_AMPS] / sum_cnt;
                if(temp >= 4 ) temp-=4;
                sum_abs_latched[CHAN_INV_AMPS] = (Word16)temp;
                temp = sum_abs[CHAN_VAC]  / sum_cnt;
                sum_abs_latched[CHAN_VAC] = (Word16)temp;
                sum_cnt=0;

                sum_abs[CHAN_LOAD_AMPS]=0;
```

```
                        sum_abs[CHAN_INV_AMPS]=0;
                        sum_abs[CHAN_VAC]=0;
                 }

                 // run the outer control
                 ControlLoop();
                }

          }
         }
         else
         {

                        // Drive both bottom switches ON if not switching

                        HWREG(PWM_BASE + PWM_GEN_1_OFFSET + PWM_O_X_GENA ) =
                                ((PWM_GEN_ACT_ZERO << PWM_GEN_ACT_A_UP_SHIFT) |
                                (PWM_GEN_ACT_ZERO <<   PWM_GEN_ACT_A_DN_SHIFT)|
                                (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_ZERO_SHIFT) |
                                (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_LOAD_SHIFT));

                        HWREG(PWM_BASE + PWM_GEN_0_OFFSET + PWM_O_X_GENA ) =
                                ((PWM_GEN_ACT_ZERO << PWM_GEN_ACT_A_UP_SHIFT) |
                                (PWM_GEN_ACT_ZERO <<   PWM_GEN_ACT_A_DN_SHIFT)|
                                (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_ZERO_SHIFT) |
                                (PWM_GEN_ACT_ZERO << PWM_GEN_ACT_LOAD_SHIFT));
        }

    //
    // Increment the clock count.
    //
        ticker++;
}


// Measure the DC offsets of the relevant AC input channels
static void MeasureOffsets(void)
{
        Word32 sums[4],mark;
        Word16 count,index;

        // reset integrators
        for(index=0;index<4;index++)
                sums[index]=0;

        // wait 2 seconds for supplies to come up
        mark=ticker;
        while((ticker - mark) < (TIMER_FREQ*2))
                WatchdogIntClear(WATCHDOG_BASE);

        // Integrate measurements to determine DC offset
        for(count=0;count<INTEGRATIONS;count++)
        {
                WatchdogIntClear(WATCHDOG_BASE);

                while(ticker == mark)  ;

                mark=ticker;

                for(index=0;index<4;index++)
                        sums[index] += ulADC[index];

        }

        // take averages and load offsets
        for(index=0;index<4;index++)
        {
                dc_offsets[index] = (Word16)(sums[index] / INTEGRATIONS);
        }
}


//*****************************************************************************
//
// Design Stellaris Contest Entry LM1633. Main line
//
//*****************************************************************************
```

```
int main(void)
{

        static char str[64];
        char chan ;
        Word32 ulData,mark;

        //
        // Set the clocking to run at 24MHz from the PLL.
        //
        SysCtlClockSet(SYSCTL_SYSDIV_10  | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_6MHZ);

        SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

        //
        // Initialize the OLED display.
        //
        OSRAMInit(false);

        //
        // Bail out if there is not a PWM peripheral on this part.
        //
        if(!SysCtlPeripheralPresent(SYSCTL_PERIPH_PWM))
        {
         OSRAMStringDraw("This part has no", 0, 0);
         OSRAMStringDraw("PWM", 36, 1);
         DiagExit(0);
        }

        //
        // Clear the screen and tell the user what is happening.
        //
        OSRAMStringDraw("Entry LM1633", 6, 0);

        //
        // Enable the peripherals used by this project
        //
        SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_WDOG);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

        //
        // Set GPIO as PWM pins.  They are used to output the PWM0, PWM1, PWM2 and PWM3
        // signals.
        //
        GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_1);
        GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

        //
        // Compute 20kHz PWM period based on the system clock.
        //
        Period = SysCtlClockGet() / TIMER_FREQ;
        // Set maximum duty cycle
        MAXIMUM_DUTY = Period-1;

        index=0;
        // set PLL to 50Hz nominal
        index_step=NOM_STEP;
        // initialise phase detector
        HWREGBITW(&g_ulFlags, FLAG_ZXPHASE)=0;
        // not locked
        HWREGBITW(&g_ulFlags, FLAG_LOCK)=0;

        // reset integrators
        sum_abs[CHAN_VAC]=sum_abs[CHAN_LOAD_AMPS]=sum_abs[CHAN_INV_AMPS]=0;
        sum_cnt=0;
        sum_abs_latched[CHAN_VAC]=0;
        sum_abs_latched[CHAN_LOAD_AMPS]=0;
        sum_abs_latched[CHAN_INV_AMPS]=0;

        // Set output voltage and stop switching
```

```
SetOutputVoltage(0);

StopSwitching();

//
// Set the PWM period
//
PWMGenConfigure(PWM_BASE, PWM_GEN_0,
            PWM_GEN_MODE_UP_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM_BASE, PWM_GEN_0, Period);

// Configure half-bridge mode with 1 us deadtime
PWMDeadBandEnable(PWM_BASE, PWM_GEN_0, 5, 5);

PWMGenConfigure(PWM_BASE, PWM_GEN_1,
            PWM_GEN_MODE_UP_DOWN | PWM_GEN_MODE_NO_SYNC);

PWMGenPeriodSet(PWM_BASE, PWM_GEN_1, Period);

// Set dead time
PWMDeadBandEnable(PWM_BASE, PWM_GEN_1, 5, 5);

// Disable fault driven output disables
PWMOutputFault(PWM_BASE, PWM_OUT_0_BIT | PWM_OUT_1_BIT |
                      PWM_OUT_2_BIT | PWM_OUT_3_BIT, false);

// Synchronise generator 0 and 1 timebases
PWMSyncTimeBase(PWM_BASE, PWM_GEN_0_BIT | PWM_GEN_1_BIT);

// Set up ADC triggering at midpoint of centred PWM output
PWMGenIntTrigEnable(PWM_BASE, PWM_GEN_0, PWM_TR_CNT_LOAD);

//
// Set PWM to a duty cycle of 0%
//
PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, 20);
PWMPulseWidthSet(PWM_BASE, PWM_OUT_2, 20);


//
// Enable the PWM output signals.
//
PWMOutputState(PWM_BASE, PWM_OUT_0_BIT | PWM_OUT_1_BIT |
                      PWM_OUT_2_BIT | PWM_OUT_3_BIT, true);

//
// Enable the PWM generator.
//
PWMGenEnable(PWM_BASE, PWM_GEN_0);
PWMGenEnable(PWM_BASE, PWM_GEN_1);

//
// Set the period of the watchdog timer.
//
WatchdogReloadSet(WATCHDOG_BASE, SysCtlClockGet());

//
// Enable reset generation from the watchdog timer.
 //
WatchdogResetEnable(WATCHDOG_BASE);

//
// Enable the watchdog timer.
//
WatchdogEnable(WATCHDOG_BASE);

//
// Configure the ADC to sample the input channels when the PWM period expires.
// After sampling, the ADC will interrupt the processor
//
ADCSequenceConfigure(ADC_BASE, 0, ADC_TRIGGER_PWM0, 0);
ADCSequenceStepConfigure(ADC_BASE, 0, 0,
                      ADC_CTL_CH0 );
ADCSequenceStepConfigure(ADC_BASE, 0, 1,
                      ADC_CTL_CH1 );
ADCSequenceStepConfigure(ADC_BASE, 0, 2,
                      ADC_CTL_CH2 );
```

```
ADCSequenceStepConfigure(ADC_BASE, 0, 3,
                         ADC_CTL_CH3 | ADC_CTL_IE | ADC_CTL_END);
ADCSequenceEnable(ADC_BASE, 0);
ADCIntEnable(ADC_BASE, 0);
IntEnable(INT_ADC0);


//
// Configure the timer1B to capture the mains line period and handle overflows
// with timer1A
//
TimerDisable(TIMER1_BASE, TIMER_B);
TimerConfigure(TIMER1_BASE, TIMER_CFG_16_BIT_PAIR | TIMER_CFG_B_CAP_TIME |
                            TIMER_CFG_A_PERIODIC);
TimerControlEvent(TIMER1_BASE, TIMER_B, TIMER_EVENT_POS_EDGE );
TimerLoadSet(TIMER1_BASE, TIMER_A, 0xFFFF);
IntEnable(INT_TIMER1B);
IntEnable(INT_TIMER1A);
TimerIntEnable(TIMER1_BASE, TIMER_CAPB_EVENT | TIMER_TIMA_TIMEOUT );
TimerEnable(TIMER1_BASE, TIMER_BOTH);

// Set I/O pin states
GPIODirModeSet(GPIO_PORTC_BASE, MAINS_ZX, GPIO_DIR_MODE_HW);
GPIODirModeSet(GPIO_PORTC_BASE, PUSH_BUTTON, GPIO_DIR_MODE_IN);
GPIOPinWrite(GPIO_PORTC_BASE, USER_LED, 0);
GPIODirModeSet(GPIO_PORTC_BASE, USER_LED, GPIO_DIR_MODE_OUT);

// Open relay
SetRelay(0);

// Measure DC offsets
MeasureOffsets();

// Start the inverter controller
StartSwitching();

//
// Throw away any button presses that may have occurred
//
HWREGBITW(&g_ulFlags, FLAG_BUTTON_PRESS) = 0;

// Reset channel display number
chan=0;

// Reset timer
mark=ticker;

//
// Loop forever while the inverter operates
//
while(1)
{

// Clear the watchdog
WatchdogIntClear(WATCHDOG_BASE);

       //
// Read the push button.
//
ulData = GPIOPinRead(GPIO_PORTC_BASE, PUSH_BUTTON) ? 1 : 0;

//
// See if the push button state doesn't match the debounced push button
// state.
//
if(ulData != HWREGBITW(&g_ulFlags, FLAG_BUTTON))
{
 //
 // Increment the debounce counter.
 //
 HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_LOW) ^= 1;
 if(!HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_LOW))
 {
     HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_HIGH) = 1;
 }

 //
```

```
// See if the debounce counter has reached three.
//
if(HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_LOW) &&
               HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_HIGH))
{
    //
    // The button has been in the new state for three consecutive
    // samples, so it has been debounced.  Toggle the debounced state
    // of the button.
    //
    HWREGBITW(&g_ulFlags, FLAG_BUTTON) ^= 1;

       //
       // If the button was just pressed, set the flag to indicate that
       // fact.
       //
       if(HWREGBITW(&g_ulFlags, FLAG_BUTTON) == 0)
       {
        HWREGBITW(&g_ulFlags, FLAG_BUTTON_PRESS) = 1;
       }
       }
}
else
{
       //
       // Since the button state matches the debounced state, reset the
       // debounce counter.
       //
       HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_LOW) = 0;
       HWREGBITW(&g_ulFlags, FLAG_DEBOUNCE_HIGH) = 0;
}

if(HWREGBITW(&g_ulFlags, FLAG_BUTTON) != 0)
{
               // process button press here

}

// See if its time to update the LCD (twice per second)
if((ticker - mark) > TIMER_FREQ/2)
{
          mark=ticker;

          // Generate the top status line
          str[0]=0;

          if(HWREGBITW(&g_ulFlags, FLAG_RELAY))
               // source if pv
               strcat(str,"Sola ");
          else
               // source is grid
               strcat(str,"Grid ");

           if(HWREGBITW(&g_ulFlags, FLAG_LOCK))
               // locked to grid
               strcat(str,"Lck ");
           else
               // syncing up to grid
               strcat(str,"Syn ");

           if(HWREGBITW(&g_ulFlags, FLAG_RATED))
               // running at rated voltage
               strcat(str,"Vo ");
           else
               // running at reduced voltage
               strcat(str,"Vr ");

           if(HWREGBITW(&g_ulFlags, FLAG_LOAD_HIGH))
               // pv output is too low
               strcat(str,"OL");
           else
               // pv output is ok
               strcat(str,"OK");

               OSRAMStringDraw(str,6,0);

            // Display a channel on the bottom line of the display
```

```
                    switch(chan)
                        {

                                case CHAN_VDC:
                                        ulData = (Word32)(ulADC[CHAN_VDC] * VDC_FULLSCALE * 10) >> 10;

                                        sprintf(str,"Vpv %lu.%lu V", ulData/10, ulData%10);
                                        break;

                            case CHAN_VAC:

                                        ulData =(Word32)(sum_abs_latched[CHAN_VAC] *
                                                 (Word32)(ABS2RMS * 2560.0 * VAC_FULLSCALE)) >> 17;

                                        sprintf(str,"Vo  %lu.%lu V", ulData/10, ulData%10);
                                        break;

                            case CHAN_LOAD_AMPS:

                                        ulData = (Word32)(sum_abs_latched[CHAN_LOAD_AMPS] *
                                                 (Word16)(ABS2RMS * 2560.0 * IDC_FULLSCALE)) >> 17;

                                        sprintf(str,"Iac %lu.%lu A", ulData/10, ulData%10);
                                        break;

                            case CHAN_INV_AMPS:

                                        ulData = (Word32)(sum_abs_latched[CHAN_INV_AMPS] *
                                                 (Word16)(ABS2RMS * 2560.0 * IDC_FULLSCALE)) >> 17;

                                        sprintf(str,"Io  %lu.%lu A", ulData/10, ulData%10);
                                        break;

                                case CHAN_FREQ:

                                        ulData = (index_step * ((TIMER_FREQ * 10)>>4)) >> (25-4);

                                        sprintf(str,"Fo  %lu.%lu Hz", ulData/10, ulData%10);
                                        break;

                                case CHAN_MAINS:

                                        if(mains_zx)
                                        {
                                                 mains_zx=0;

                                                 ulData = (Word32)(200000000Lu / zx_period);

                                                 if(ulData > 450 && ulData < 650)
                                                 sprintf(str,"Fac %lu.%lu Hz", ulData/10, ulData%10);
                                                 else
                                                 {

                                                           HWREGBITW(&g_ulFlags, FLAG_LOCK)=0;
                                                           pll_cnt=128;
                                                           index_step=NOM_STEP;


                                                           sprintf(str,"Fac ?     ");
                                                 }
                                        }
                                        else
                                        {

                                                 HWREGBITW(&g_ulFlags, FLAG_LOCK)=0;
                                                 pll_cnt=128;
                                                 index_step=NOM_STEP;


                                                 sprintf(str,"Fac ?     ");
                                        }

                                        break;

                        }

                        // advance to next channel
                        if(++chan >= 6)
                                chan=0;


                        OSRAMStringDraw("             ",6,1);

                        OSRAMStringDraw(str, 6, 1);

                }
        }
}
```

SINLUT.C

```
//
// Normalised sine wave lookup table. 512 entries corresponds to 360 degrees
// Sine values are stored in signed 15 bit fixed point format.
//
const signed short sinlut[512]= {

0,402,804,1206,1607,2009,2410,2811,3211,3611,4011,4409,4807,5205,5601,5997,
6392,6786,7179,7571,7961,8351,8739,9126,9511,9895,10278,10659,11038,11416,11792,12166,
12539,12909,13278,13645,14009,14372,14732,15090,15446,15799,16150,16499,16845,17189,
17530,17868,18204,18537,18867,19194,19519,19840,20159,20474,20787,21096,21402,21705,
22004,22301,22594,22883,23169,23452,23731,24006,24278,24546,24811,25072,25329,25582,
25831,26077,26318,26556,26789,27019,27244,27466,27683,27896,28105,28309,28510,28706,
28897,29085,29268,29446,29621,29790,29955,30116,30272,30424,30571,30713,30851,30984,
31113,31236,31356,31470,31580,31684,31785,31880,31970,32056,32137,32213,32284,32350,
32412,32468,32520,32567,32609,32646,32678,32705,32727,32744,32757,32764,32767,32764,
32757,32744,32727,32705,32678,32646,32609,32567,32520,32468,32412,32350,32284,32213,
32137,32056,31970,31880,31785,31684,31580,31470,31356,31236,31113,30984,30851,30713,
30571,30424,30272,30116,29955,29790,29621,29446,29268,29085,28897,28706,28510,28309,
28105,27896,27683,27466,27244,27019,26789,26556,26318,26077,25831,25582,25329,25072,
24811,24546,24278,24006,23731,23452,23169,22883,22594,22301,22004,21705,21402,21096,
20787,20474,20159,19840,19519,19194,18867,18537,18204,17868,17530,17189,16845,16499,
16150,15799,15446,15090,14732,14372,14009,13645,13278,12909,12539,12166,11792,11416,
11038,10659,10278,9895,9511,9126,8739,8351,7961,7571,7179,6786,6392,5997,5601,5205,
4807,4409,4011,3611,3211,2811,2410,2009,1607,1206,804,402,0,-402,-804,-1206,-1607,
-2009,-2410,-2811,-3211,-3611,-4011,-4409,-4807,-5205,-5601,-5997,-6392,-6786,-7179,
-7571,-7961,-8351,-8739,-9126,-9511,-9895,-10278,-10659,-11038,-11416,-11792,-12166,
-12539,-12909,-13278,-13645,-14009,-14372,-14732,-15090,-15446,-15799,-16150,-16499,
-16845,-17189,-17530,-17868,-18204,-18537,-18867,-19194,-19519,-19840,-20159,-20474,
-20787,-21096,-21402,-21705,-22004,-22301,-22594,-22883,-23169,-23452,-23731,-24006,
-24278,-24546,-24811,-25072,-25329,-25582,-25831,-26077,-26318,-26556,-26789,-27019,
-27244,-27466,-27683,-27896,-28105,-28309,-28510,-28706,-28897,-29085,-29268,-29446,
-29621,-29790,-29955,-30116,-30272,-30424,-30571,-30713,-30851,-30984,-31113,-31236,
-31356,-31470,-31580,-31684,-31785,-31880,-31970,-32056,-32137,-32213,-32284,-32350,
-32412,-32468,-32520,-32567,-32609,-32646,-32678,-32705,-32727,-32744,-32757,-32764,
-32767,-32764,-32757,-32744,-32727,-32705,-32678,-32646,-32609,-32567,-32520,-32468,
-32412,-32350,-32284,-32213,-32137,-32056,-31970,-31880,-31785,-31684,-31580,-31470,
-31356,-31236,-31113,-30984,-30851,-30713,-30571,-30424,-30272,-30116,-29955,-29790,
-29621,-29446,-29268,-29085,-28897,-28706,-28510,-28309,-28105,-27896,-27683,-27466,
-27244,-27019,-26789,-26556,-26318,-26077,-25831,-25582,-25329,-25072,-24811,-24546,
-24278,-24006,-23731,-23452,-23169,-22883,-22594,-22301,-22004,-21705,-21402,-21096,
-20787,-20474,-20159,-19840,-19519,-19194,-18867,-18537,-18204,-17868,-17530,-17189,
-16845,-16499,-16150,-15799,-15446,-15090,-14732,-14372,-14009,-13645,-13278,-12909,
-12539,-12166,-11792,-11416,-11038,-10659,-10278,-9895,-9511,-9126,-8739,-8351,-7961,
-7571,-7179,-6786,-6392,-5997,-5601,-5205,-4807,-4409,-4011,-3611,-3211,-2811,-2410,
-2009,-1607,-1206,-804,-402

};
```