Real Time Implementation of a Software PLL on a TMS320C6713 DSP

Ayoub Bengherbia*, Noureddine Batel*
*Laboratory of Advanced Electronic Systems (LSEA), Department of Electrical Engineering, Faculty of Technology, University of Yahia Fares, Medea 26000 Algeria; (e-mail:bengherbia.ayoub@univ-medea.dz, batel.noureddine@univ-medea.dz)

Abstract-
This paper discusses the design and analysis of Software Phase Locked Loop (SPLL). The study of phase-locked loops (PLLs) has been extensively covered in the literature, and most of its theoretical and analytical results are verified by simulations on a PSpice (16.6 version). The SPLL algorithm is implemented on a 32-bit floating point Digital Signal Processor (DSP), using a C6713 simulator from Texas Instruments. The most of the program is shared between the acquisition of the input signal using two external hardware interrupt routines, the SPLL processing, and the graphical representation of the results on Matlab (7.1 version) based on a Real Time Data Exchange (RTDX). Some results are analyzed and validate the associated theoretical results on the implemented system. The gradual optimization of the code, going through C, then linear assembly, to accomplish our work with pure assembly, allows real-time SPLL processing under increasingly high input frequencies.

Index terms—**Phase Locked Loop (PLL), Software Phase Locked Loop (SPLL), Digital Signal Processor (DSP), Code Composer Studio (CCS), Texas Instruments (TMS), Real Time Data Exchange (RTDX).**
DOI Number: 10.48047/nq.2023.21.6.NQ23160 NeuroQuantology2023;21(6): 1586-1600

I. INTRODUCTION

The PLL is among the most widespread electronic components. Present in the majority of electronic devices ranging from televisions to mobile phones, it has a wide range of applications including frequency synthesis, clock recovery, carrier restitution, signal synchronization, modulation-demodulation angular, The speed control of the engines etc ... [1], to name only a few relating to the fields of communications, electronic instrumentation, and automatic.

In the 1980s, the dominance in phase-locked circuits was that of analog or linear PLLs (LPLL) as well as semi-digital PLLs (DPLL). In the last decade, two different types of PLL have gained importance: digital PLLs (ADPLL) and software PLLs (SPLL). The ADPLL built entirely by logic circuits, eliminates the disadvantages of the DPLL related to its analog part which suffers from the problems relating to the variations or drifts of the components [1]. The current existence of fast DSPs has greatly opened the field to SPLL applications. These can be programmed to act as an LPLL, a DPLL, or an ADPLL. Thus, the SPLL type is the most universal of the PLLs, although it must fulfill a necessary condition: the calculation algorithm fulfilling the function of a PLL must imperatively be executed at least once at each period of the input signal, thus severely limiting the range covered in frequency [1].

Very recently [2], the neuronal PLL (NPLL) is developing. In this new type of PLL, all of its constituents are based on neural networks, including its rate-controlled oscillator (RCO). NPLL is used in the tracking and decoding of low frequency modulations (<1 kHz), such as those that occur in biological active sensors in mammals. Note, however, that the different types of PLLs just mentioned act differently and that there is no common theory that covers them all [1].

The remainder of the paper is organized as follow: In the second part, a design is carried out with the help of Pspice in order to achieve an optimum DPLL in both static and dynamic behavior. It includes a Detector Frequency-Phase (DFP), which is regarded as one of the most complex detectors and, when coupled with the loop filter, forms a charge pump with extended, if not infinite, operating ranges. The Voltage-Controlled Oscillator (VCO) is modified so that its voltage-

frequency characteristic is linearized. The temporal constants of the loop filter are assumed to be optimal.

A third part is reserved for the design of the SPLL, it is entrusted with the aspect of the DPLL from algorithms performing the three fundamental functions, that of the nonlinear DFP, the filter optimal looping, and VCO with quasi-linear voltage-frequency characteristic. The fourth part summarizes the results of this work and draws conclusions.

II. DPLL DESIGN

Classic DPLL is a hybrid system built from analog and digital blocks. The block diagram is shown in figure 1.

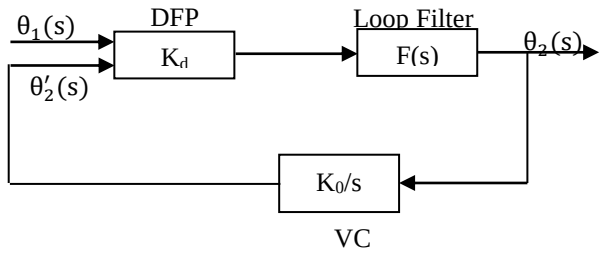


Fig. 1. DPLL block diagram.

It consists of three functional blocks which are, the Phase Detector Freauency (DFP), the Loop Filter (LF), and the Voltage Controlled Oscillator (VCO). The only really digital part is the phase detector. In terms of operating principle, the phase locking loop is used to synchronize the phase and output frequency of the controlled oscillator to match the phase and frequency of an input signal [1].

II.1 Phase Frequency Detector (DFP)

Detector Frequency Phase (DFP) figure 2, presents as its name suggests, a signal at its output which does not depend only on the phase error but also on the frequency error $\Delta\omega = \omega_1-\omega_2$ when the loop is not yet locked [1]. The DFP is designed from two D flip-flop whose outings are noted "UP" and "DN", respectively. By an additional AND logic gate deleting the UP output state UP= Dn=1 by acting on the CLEAR inputs of the two tips, the DFP can be in one of the three states that remain (denoted –1, 0, 1) [3] [16]:

UP = 0, DN=1 : state –1,
UP = 0, DN=0 : state 0,
UP = 1, DN=0 : state +1.

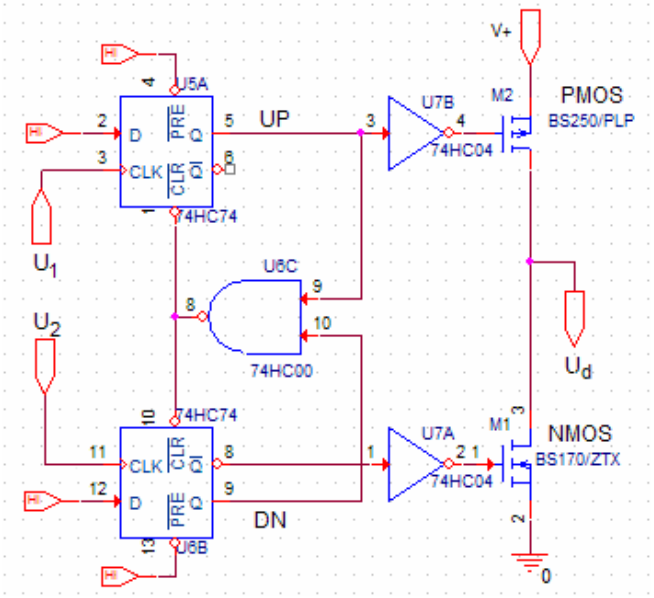


Fig. 2. Detector Frequency Phase (DFP).

The current state of the DFP is determined by the positive transitions of the U_1 and U_2 input signals as explained on the state diagram in figure 3 [1]. Thus, an upright front of U_1 forces the DFP to go to the next higher state, unless it is already in the state +1. By analogy, an upright front of U_2 forces the DFP to go to the next lower state, unless it is already in the state –1[14].

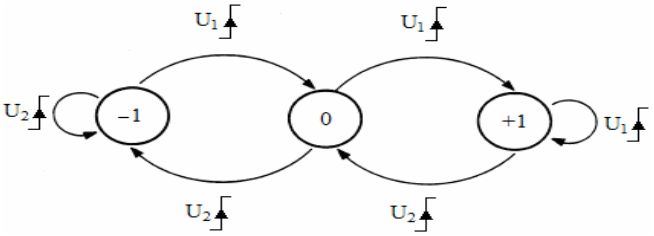


Fig. 3. DFP state diagram

II.2 Loop filter

In general, the output of the phase comparator contains unwanted terms often located in the upper part of the spectrum relatively the useful term of detection. These unnecessary components are filtered by the loop filter, which is therefore necessarily of the low-pass type. In most PLL designs a first-order pass filter is used, whose figure 4 gives the frequently encountered version [1][3-4]. This filter is in advance phase retard with a pole and zero. An in advance phase network whose action comes from zero (numerator from the filter transfer function), is therefore combined with another phase delay whose action comes from the pole (denominator of the transfer function of the filter)[15].

The passive filter, figure 4, has the transfer function:

$$F(s) = \frac{1 + \tau_2 s}{1 + (\tau_1 + \tau_2)s} \quad (1)$$

Where $\tau_1 = R_1C$ and $\tau_2 = R_2C$.

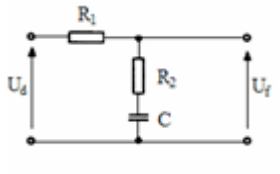


Fig. 4. Low-pass filter.

However, a special behavior of the loop filter manifests in the presence of the DFP type phase detector. The latter provides no current circulation in resistors R_1 and R_2 when it is in the high impedance state. Under this condition, the voltage across the capacitor remains theoretically unchanged. By neglecting the leakage currents, the U_F output signal of the filter has a constant level when the DFP is in its state 0. Thus, the filter acts as an ideal integrator, and its transfer function becomes [1]:

$$F(s) = \frac{1 + \tau_2 s}{(\tau_1 + \tau_2)s} \quad (2)$$

The combination of the DFP with the passive filter, is often referenced as a charge pump. This means that in the high state of the DFP output signal, the charge circulates towards the capacitor of the filter in other words, the charge is pumped inside the capacitor. When the exit signal of DFP is in its low state, the charge circulates from the filter capacitor, that is the charge is pumped outside the capacitor towards the mass. The pumping stops, however, during state 0 of the DFP. Therefore, we retain for the design of the DPLL, the charge pump made up of the DFP detector associated with the passive filter in figure 4.

In order not to dampen the output of the filter, a matching circuit in the first stage of the VCO is provided. The filter signal is therefore BF and a simple operational amplifier of the 741 type mounted as a follower suffices as an adapter stage, figure 5 [16].

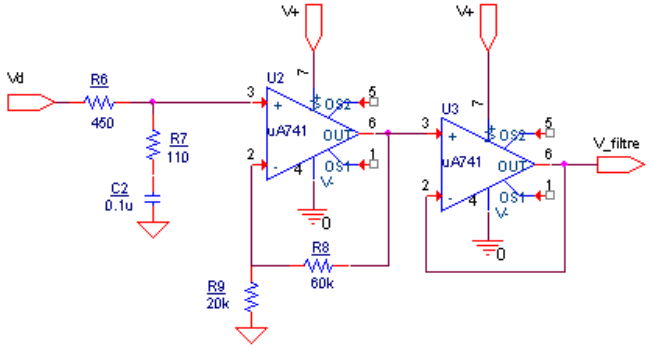


Fig. 5. The loop filter

II.3 Voltage Controlled Oscillator

The VCO used in figure 1, is the second analog part of the DPLL after the loop filter [1]. The VCO oscillates at the angular frequency ω_2 which is determined by the U_F output signal of the loop filter:

$$\omega_2(t) = \omega_0 + K_0 U_f(t) \quad (3)$$

Where ω_o is the central frequency of the VCO and K_0 is the sensitivity or gain of the VCO, in $(SV)^{-1}$. When the VCO model uses the output phase θ_2 and not the output frequency ω_2, the function of VCO transfer expressed by the Laplace Transform is written:

$$\frac{\theta_2(s)}{U_f(s)} = \frac{K_0}{s} \quad (4)$$

For phase signals, a VCO therefore simply represents an integrator, while for frequency signals, a phase detector simply becomes an integrator.

The characteristic of the VCO is to be determined from the central frequency ω_0 or its row, which must be known. In most cases, the VCO is part of the DPLL which is typically an integrated circuit. Data-Sheet indicates the $u_{f\,min} \leq u_F \leq u_{fmax}$ used for control voltage (typically, 1 to 4V for a 5V power supply). The K_0 sensitivity of the VCO is calculated as:

$$K_0 = \frac{\omega_{2max} - \omega_{2min}}{u_{f\,min} - u_{f\,min}} \quad (5)$$

The external components of the VCO can also be determined, generally, by indication of the data-sheet. We prefer to carry out a design of the VCO, according to the principle [3] in figure 6:

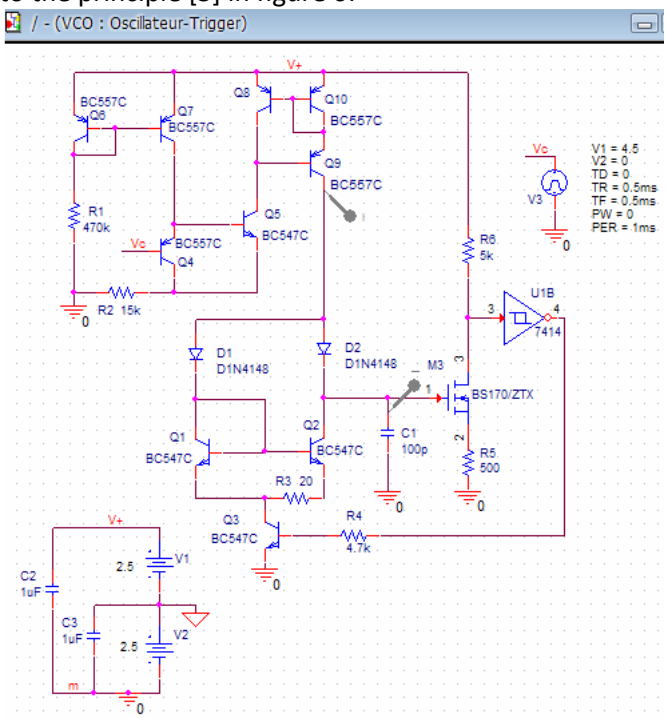


Fig. 6. VCO

The characteristic of the VCO obtained in figure 7 is almost linear on the control rang, V_f = 0.1V to 3.6V, for

which the frequency varies from 50 kHz to 967 kHz (exceeding the decade by about one octave) [16].

The average slope of the characteristic is measured by the tangent in the middle point of the curve (V_f = 1.8V, f = 585.6 kHz). Or, an average sensitivity of the VCO: K_0 = 325 kHz/V = 2.10^6 rd/Vs.

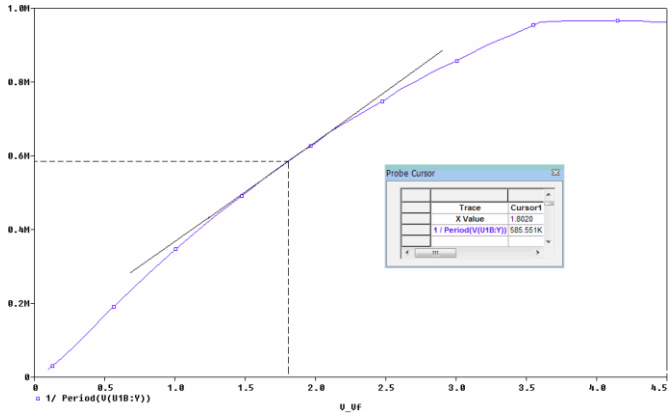


Fig. 7. VCO Frequency-Tension characteristic

II.4 Operation of a DPLL

When the DPLL is in the locked state, its performance can be analyzed by a linear model. The figure 1[1] shows the mathematical model of the locked DPLL, from which we can calculate the phase transfer function H(s) which connects the phase θ_1 of the input signal to the phase θ_2 of the output signal:

$$H(s) = \frac{\theta_2(s)}{\theta_1(s)} = \frac{K_0 K_d F(s)}{s + K_0 K_d F(s)} = \frac{\omega_2(s)}{\omega_1(s)} \qquad (6)$$

Considering the transfer function F(s) of the passive loop filter (equation 1), the transmittance H(s) of the DPLL has a second-order denominator that it is more convenient to put in normalized form, $s^2 + 2\varepsilon\omega_n s + \omega_n^2$ where ω_n represents the natural frequency of the DPLL, and ε, its damping factor. So we have:

$$H(s) = \frac{\frac{K_0 K_d}{\tau_1 + \tau_2}(1 + \tau_2 s)}{s^2 + \frac{1 + K_0 K_d \tau_2}{(\tau_1 + \tau_2)} s + \frac{K_0 K_d}{N(\tau_1 + \tau_2)}} \qquad (7)$$

Where:

$$\omega_n = \sqrt{\frac{K_0 K_d}{(\tau_1 + \tau_2)}} \text{ and } \quad \varepsilon = \frac{\omega_n}{2}\left(\frac{1}{K_0 K_d} + \tau_2\right) \qquad (8)$$

The product $K=K_0K_d$ (with F(0)=1) has dimension S^{-1} and represents the open loop static gain. The equation represents the second-order DPLL system, which acts as a low-pass filter for the input signals $\theta_1(t)$ and its transfer function becomes:

$$H(s) = \frac{\omega_n^2(1 + \tau_2 s)}{s^2 + 2\varepsilon\omega_n s + \omega_n^2} \qquad (9)$$

The three functional blocks of the DPLL, of figures 2, 5, and 6, are assembled in a closed loop, as shown in figure 8[16].

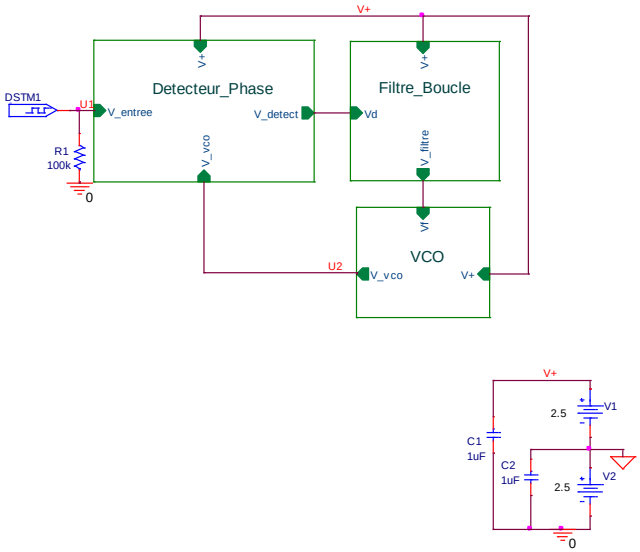


Fig. 8. Block diagram of the DPLL simulated

III. SPLL DESIGN

When the input signal U_1 of a PLL is a binary signal, it is more appropriate to implement a SPLL to operate as a DPLL [14]. In our case, the SPLL algorithm will use the functions of a Detector Frequency-Phase (DFP) in figure 2, associated with the passive filter in figure 5, or in all, a load pump. The required functions are represented by the block diagram of the figure 9[1]. This consists of three functional blocks: a DFP algorithm, a filtering algorithm, and a controlled phase oscillation algorithm :

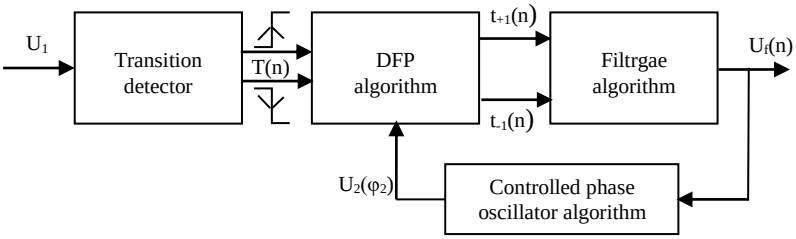


Fig. 9. Block diagram of the operations of a SPLL operating as a DPLL

The filter algorithm must operate in the same way as the passive filter of the DPLL, figure 5. The only signal that exists physically is the $u_1(t)$ input signal with a square shape, the frequency of which can vary in the frequency range of the programmed VCO. The $u_2(t)$ fictitious signal from the SPLL must essentially provide information on its phase $\phi_2(t)$.

From figure 3, we know that the logical state of the DFP depends on the positive transitions of its two entry signals, u_1 and u_2.

As we need to know the moments of switching the positive fronts of $u_1(t)$ and $u_2(t)$, we use the transitions (positive and negative) of $u_1(t)$ in order to generate requests of interruption, figure 9. As soon as the interruption is recognized, its moment of appearance is safeguarded. The DSP used must contain a timer for this. The moments of interruption detection are called t(0), t(1),…, t(n),…, figure 10. Before discussing the SPLL algorithm, a certain number of signals must be defined [1], figure 10.

- $u_1(n)$ is the sampled version of the continuous reference signal $u_1(t)$ immediately after occurrence of the interruption request. Either at time t(n-1), $u_1(n) = 1$, and time t(n), $u_1(n) = 0$.
- $\phi_2(t)$ is the fictitious continuous phase of the oscillator signal. It evolves linearly over time. $\phi_2(n)$ is its sampled version. Likewise, the samples are taken from the moments of interruption.
- $u_2(t)$ is the fictitious continuous signal of the oscillator, which will be calculated from the phase $\phi_2(t)$. When ϕ_2 in its growth goes through 0 (or 2π), u_2 generates its positive front. While when in its growth, ϕ_2 goes through π, u_2 generates its negative front.
- $u_d(t)$ is the fictitious continuous output of the signal or state of the DFP, which can have the values −1, 0, or 1. as explained in figure 3, the updated front of u_1 forces the DFP to go to the next Higher state, unless it is already in the state +1, while the upright front of u_2 forces the DFP to go to the next lower state, unless it is already in the State –1. $u_d(n)$ is the sampled version of $u_d(t)$ and is defined as DFP condition just before the occurrence of the interruption. For example, $u_d(n-1)$ has value 0, because $u_d(t)$ was in state 0 before the interruption was triggered at t=t(n-1).
- We also define a t_{cross} crossing time by the zero value of the phase ϕ_2, relative to the previous transition of u_1.
- T(n) is defined as a time interval, t(n) - t(n-1), that is to say between that of the most recent interruption and that of the previous interruption. When $u_d(t)$ is in the state +1 in a fraction of the interval T(n), the corresponding duration with T_{cross} is stored in the variable $T_{+1}(n)$, figure 10. When $u_d(t)$ is in the state −1 in a fraction of the interval T(n), the complementary duration of T_{cross} to T(n) is stored this time in the variable $T_{-1}(n)$, figure 10.
- $u_c(n)$ is used to describe the sampled signal at the fictitious capacity of the loop filter in figure 4.
- $u_f(n)$ is used to describe the sampled signal at the outlet of the loop filter in figure 9 (which must respond as the analog filter in figure 4). $u_f(n)$ is identical to $u_c(n)$ except when the ''current'' travels the fictitious capacity of the filter.

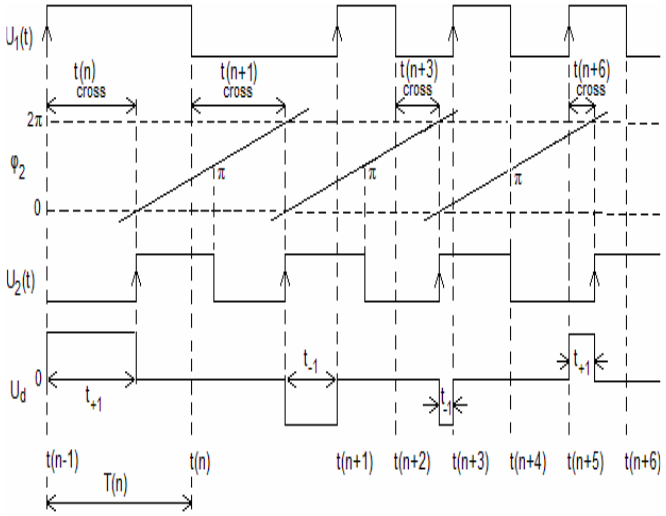


Fig. 10. Plot of the signals calculated by the SPLL algorithm in the presence of the U_1 input

The flowchart of figure 11 shows what to do with each interruption service [1]. The signals appearing in this flowchart are illustrated in figure 10. The highest portion of the SPLL algorithm is trivial and lists the initialization of certain variables, classified according to the order of their use. The program enters an endless loop where it awaits the arrival of a next interruption. When the interruption is detected, time has elapsed from the last interruption is measured, T(n)=t(n)-t(n-1). The current value of the reference signal $u_1(t)$ is then deduced, $u_1(n)=u_1(t)$, determining the presence of the positive or negative half of the signal $u_1(t)$. The oscillator is supposed to generate a form of square wave $u_2(t)$, which must be calculated indirectly from its phase $\phi_2(t)$ whose instant angular frequency is given by:

$$\omega_2(t) = K_0 u_f(t) \quad (10)$$

Where initially, $\omega_2(0) = K_0 u_f(0)$.

Using the Walsh function, the continuous output signal $u_2(t)$ therefore becomes:

$$u_2(t) = W[\phi_2(t)] \quad (11)$$

With :

$$\phi_2(t) = \int \omega_2(t)dt \quad (12)$$

We assume that the current time t is t(n) which corresponds to the second interrupt request as shown in figure 10, but we do not know the value of the phase $\phi_2(t)$ at this time. However, we can determine the angular step from time t(n−1), since the value of the filter output signal $u_f(n-1)$ can be calculated and therefore we also know the instantaneous angular frequency $\omega_2(n-1)$ of the SPLL controlled oscillator.

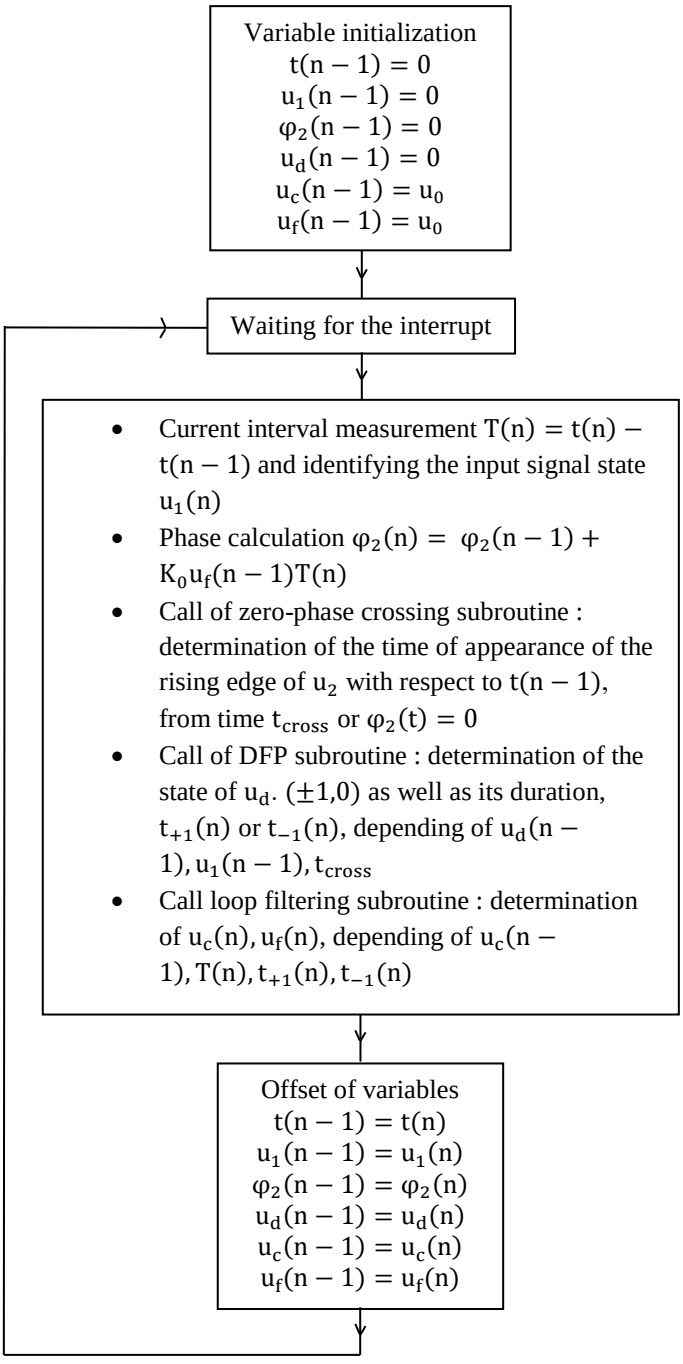


Fig. 11. Flowchart of the operations performed by the SPLL algorithm

Assuming that $u_f(n-1)$ remains constant during the interval $t(n-1) \le t < t(n)$, the phase of the oscillator must change by an increment equal to:

$$\Delta\phi_2 = K_0 u_f(n-1)T(n) \quad (13)$$

Thus, at time t = t(n) we have:

$$\phi_2(n) = \phi_2(n-1) + K_0 u_f(n-1)T(n) \quad (14)$$

Note that the phase $\phi_2(n-1)$ at time t=t(n−1) is known, because the phase signal is computed recursively with the initial value $\phi_2(0)=0$ at t=0.

We also note that in figure 10, the shape of $\phi_2(t)$ is theoretically represented by straight line segments which are the average tendencies of the general shape of $\phi_2(t)$.

In the rest of the flowchart of figure 11, three subroutines are called successively before updating the initialization variables and returning to the state of waiting for a new interrupt.

The subroutines mentioned in figure 11 are three in number. The flowcharts for the zero-phase crossover routines in figure 12, the DFP in figure 14, and the loop filter in figure 15, are shown separately.

III.1 Zero-phase crossover flowchart

The flowchart in figure 12 is responsible for determining whether the fictitious signal $u_2(t)$ has shown a positive edge in the interval T(n). This is the case when the continuous phase signal $\phi_2(t)$ crosses the value 0 (or 2π, 4π, etc.) during the interval T(n), figure 10. To prevent any arithmetic overflow in the calculations, we periodically reduce the total phase by 2π when it exceeds 2π. When the phase crosses such a limit, the corresponding instant is calculated in the variable $t_{cross}(n)$. If no crossing is detected, $t_{cross}(n)$ is set to 0.

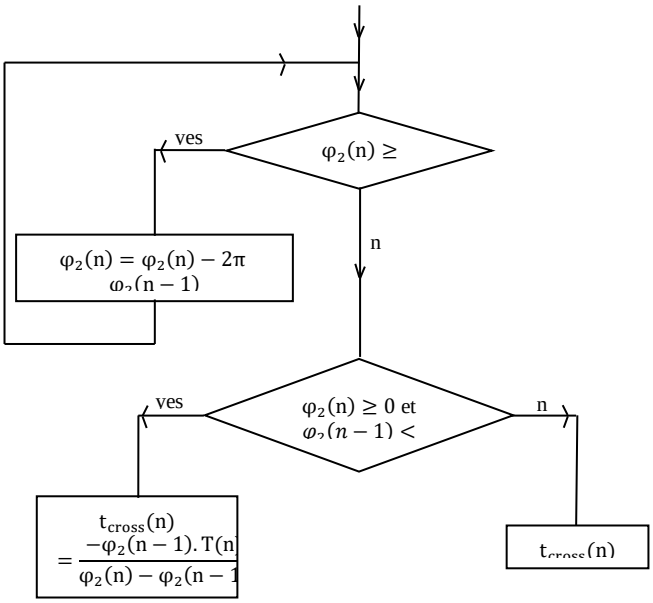


Fig. 12. Zero-phase crossing flowchart for $\phi_2(t)$

The calculation of $t_{cross}(n)$ indicated in the flowchart of figure 12, is based on the geometric observation of figure 13 taken in part from figure 10. By symmetry of the two triangles of figure 13, we get:

$$\operatorname{cotg}(\delta) = \frac{t_{cross}}{-\phi_2(n-1)} = \frac{T(n)}{\phi_2(n) - \phi_2(n-1)} \qquad (15)$$

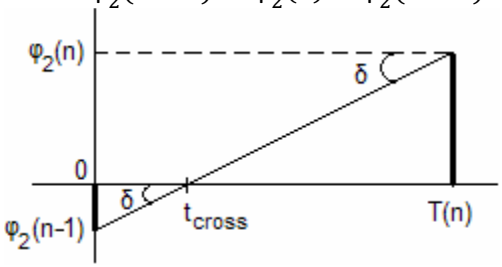


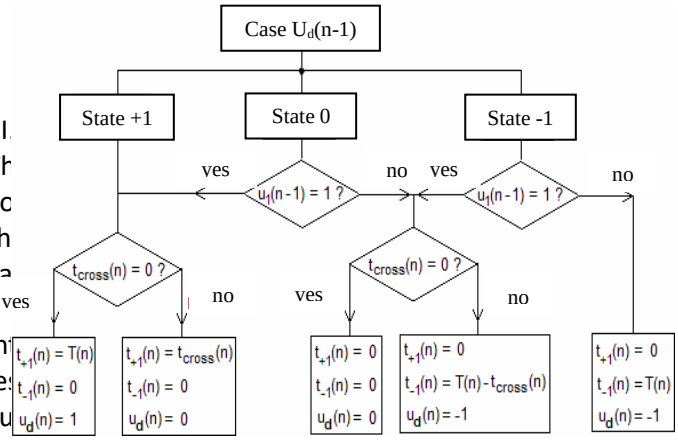
Fig. 13. Geometric determination of t_{cross}

III.2 DFP flowchart

The flowchart of figure 14 now becomes ready to calculate the state $U_d(t)$ of the DFP during the interval T(n) and save its duration in $t_{\pm1}(n)$ at time t(n), figure 10. In addition to t_{cross} giving information on the rising edge of $u_2(t)$, the signal $U_d(t)$ also depends on the rising edge of $u_1(t)$ which is detected when $u_1(n-1) = 1$, figures 10 and 14. Similarly, the state of $U_d(n-1)$ just before time $t = t(n-1)$ must be known. Thus if as in figure 10, $U_d(n-1) = 0$ and $u_1(t)$ marks a positive transition at $t = t(n-1)$, $U_d(t)$ goes to state +1. When $u_2(t)$ subsequently marks a positive transition, $U_d(t)$ returns to state 0. When $u_2(t)$ again marks a positive transition, $U_d(t)$ goes to state −1. $U_d(t)$ returns to state 0 following a positive transition from $u_1(t)$ to $t = t(n+1)$.

Everything is therefore decided according to the DFP state diagram of figure 3, and the algorithm of figure 14 shows nine different possible cases. This algorithm determines the duration of each state ±1 in $t_{+1}(n)$ or $t_{-1}(n)$, and also calculates the state of U_d at the end of the interval T(n). This state will be used as initial condition $U_d(n-1)$ at the next interrupt service.

When it happens that $t_{+1}(n)$ is greater than 0, this means that the supply voltage must be applied during the interval $t_{+1}(n)$ to the RC filter of figure 4. When $t_{-1}(n)$ is not zero, the fictitious capacitor C must be discharged to ground during the interval $t_{-1}(n)$.



define $u_f(n)$ as the mean of $u_f(t)$ in the interval $t(n-1) \le t < t(n)$. It is also assumed that the duration of a cycle T(n) (half of the cycle duration of the reference signal) is much lower than the time constant τ_1 of the filter of figure 4. Under this condition, the current flowing in C remains constant during the charging or discharging intervals. This assumption allows simple expressions of $u_c(n)$ and $u_f(n)$ [1].

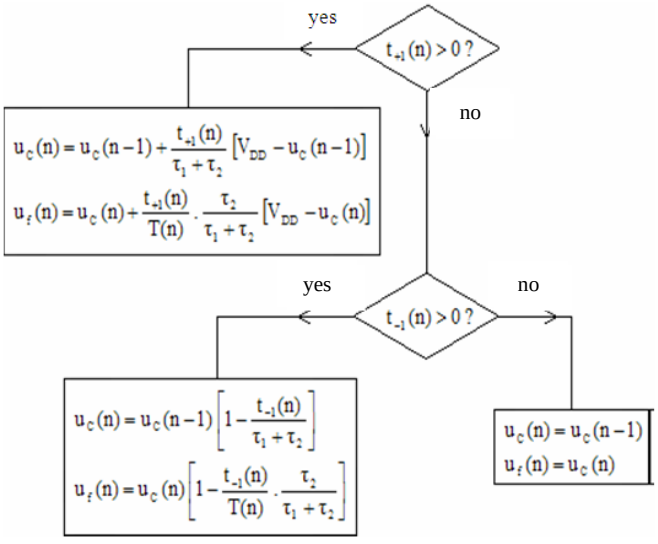


Fig. 15. SPLL loop filter flowchart

The expressions listed in figure 15 come from the following observations made in figure 4.

When $u_d(n) = +1$:

$$u_c(n) = u_c(n-1) + \Delta V_c \qquad (16)$$

With:

$$\Delta V_c = \frac{1}{C}\int_0^{t_{+1}(n)} I dt \qquad (17)$$

Where: $I = \frac{V_{DD} - u_c(n-1)}{R_1 + R_2} = c^{cst}$ (18)

$$u_f(n) = u_c(n) + R_2 I_{mean} \quad (19)$$

With: $I_{mean} = \frac{1}{T(n)} \int_0^{t_{+1}(n)} \frac{V_{DD} - u_c(n)}{R_1 + R_2} dt$ (20)

When $u_d(n) = -1$:

$$u_c(n) = u_c(n-1) e^{\frac{-t_{-1}(n)}{\tau_1 + \tau_2}} \quad (21)$$

for: $t_{-1}(n) \ll \tau_1 + \tau_2$ we have : $e^{\frac{-t_{-1}(n)}{\tau_1 + \tau_2}} = 1 + \frac{t_{-1}(n)}{\tau_1 + \tau_2}$

$$u_f(n) = u_c(n) - R_2 I_{mean} \quad (22)$$

with:

$$I_{mean} = \frac{1}{T(n)} \int_0^{t_{-1}(n)} \frac{u_c(n)}{R_1 + R_2} dt = \frac{u_c(n-1)(\tau_1 + \tau_2)}{T(n)(R_1 + R_2)} \left[1 - e^{\frac{-t_{-1}(n)}{\tau_1 + \tau_2}}\right] \quad (23)$$

That is:

$$I_{mean} = \frac{u_c(n-1) t_{-1}(n)}{T(n)(R_1 + R_2)} \quad (24)$$

Where we replace $u_c(n-1)$ by $u_c(n)$ since $u_f(n)$ must be a function of $u_c(n)$ at t=t(n).

All the calculations of a single interrupt service have just been presented. Since most of the samples computed at t=t(n) will be used as seed values in the next interrupt service, they must be time-shifted. This is shown at the bottom of the SPLL algorithm flowchart in figure 11.

IV. RESULTS AND DISCUSION

The filter responses at the 625 kHz, 290.5 kHz, 200 kHz, and 100 kHz inputs simulated on Pspice are respectively arranged from top to bottom in figure 16.

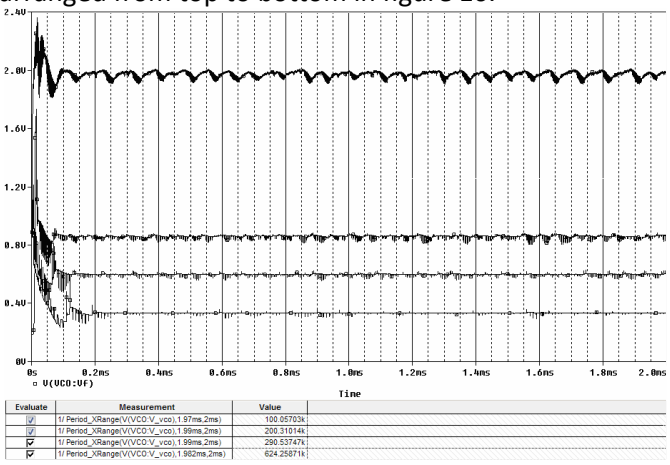


Fig. 16. Combined loop filter responses for all four input frequencies

At the bottom of this same figure, are gathered the frequencies reached by the VCO after 2 ms of simulation in response to the four control voltages from the loop filter. Small differences with input frequencies are observed. This is because once the VCO frequency happens to be equal to the input frequency, the charge pump remains in its high impedance state under which, the capacitor C_1 of the VCO (figure 6) cannot retain its charge indefinitely because of leaks. The voltage dropping until it is perceptible by the Trigger, causes a new sufficient charge of C_1 to be launched by the latter. The cycle begins again, thus varying the frequency of the VCO in a small interval around the frequency-reference input.

In the “Tools” menu of the Code Composer Studio (CCS), there are the two tools “Pin connect” and “Port connect” for the simulation of these two types of interaction, respectively.

The "Pin connect" tool allows you to simulate the external interrupts applied to the corresponding pins of the DSP. A file with a specific format can be connected to these pins. The simulated pins are of two types: pulse and waveform. Pulse pins are sensitive to rising edges of signals, while waveform pins are sensitive to levels (therefore, rising and falling edges).

In our application, the time intervals T(n) are measured, figures 10 and 11, between two successive transitions of the input signal (therefore, measurement of the time taken by each level of the signal). For this, the INT4 and INT5 pins of the pulse type are used and are therefore only sensitive to rising edges. By considering a periodic square signal at the input of the PLL, it is then enough to apply this signal to pin INT5 and its replica shifted by half a period to pin INT4 so that this meets the need. Thus, the falling edges of the input signal are transformed into rising edges at pin INT4. According to figure 17, pin INT5 is dedicated to identifying high signal levels using the hwiENTREE_ON interrupt routine, while pin INT4 is required to identify low signal levels at the using the hwiENTREE_OFF interrupt routine.

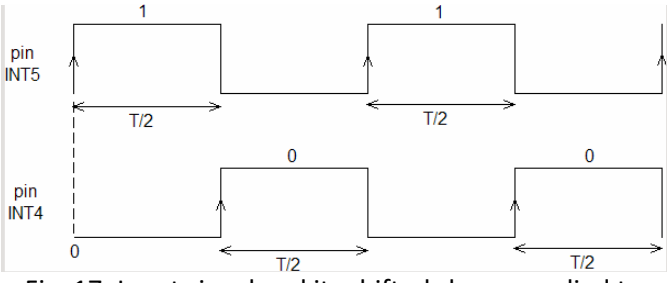


Fig. 17. Input signal and its shifted shape, applied to pins INT5 and INT4

The "Pin connect" allows you to specify the instants of occurrence of external interruptions in a simple text file that must be attached to the specified pin through a

configuration window that opens for this purpose. The text file contains the interrupt intervals in terms of CPU clock cycles [5-13], F_{CPU} = 225MHz.

Two text type input data files, FI_4.txt and FI_5.txt, each linked to an external interrupt pin, INT4 and INT5 respectively, are therefore attached to the FI frequency of the input signal in the repetitive way next :

F_I-5.txt: $\left(+\frac{F_{cpu}}{F_I}\right)$ rpt EOS

F_I-4.txt: $\frac{F_{cpu}}{2F_I}\left(+\frac{F_{cpu}}{F_I}\right)$ rpt EOS

« rpt EOS » indicates a repetition until the end of the simulation. The plus sign adds the value in clock cycles to the total value of previous cycles.

Eight files are therefore prepared for the four input frequencies of the DPLL case in figure 16, namely, 100 kHz, 200 kHz, 290 kHz, and 625 kHz.

IV.1 Real-time acquisition test of the input signal

After development of the SPLL project under CCS, the real-time SPLL application was first executed on the C6713 simulator in order to test the real-time acquisition of the input signal under the four different frequencies. The software routine performing the processing of the SPLL algorithm has been ignored, in favor of the only hardware routines measuring the signal levels and their durations T(n).

For the 100 kHz signal of half period T=5µs, figure 18 plotted by Matlab via RTDX, shows a good acquisition of information relating to the half period of the input signal.

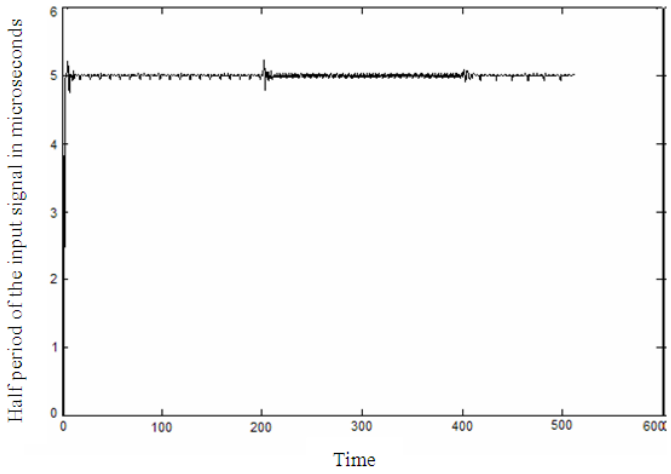


Fig. 18. Measurement of separation times between signal levels at 100 kHz

This real-time acquisition could not continue for the other higher input frequencies, probably due to certain missed interruptions during the acquisition phase. Thinking that the simulator was the source of this problem, necessary changes to this application were started in order to adapt it to a real DSP card, the DSK C6713 kit. Real interrupts emanate from a pulse generator, have been applied to pins INT4 and INT5 of the input/output port of the card. The alternation of the input bits has been detected at the 100 kHz frequency, unlike the other frequencies. It is therefore necessary to optimize the program in order to be able to satisfy its execution in real time for input frequencies greater than 100 kHz.

IV.2 Delayed acquisition of the input signal and measurement of the filtered responses

The results obtained being similar to those of the C6713 simulator, we continue the study on the simulator without, however, attaching great importance to the aspect of real-time execution as a first step. Therefore, delayed-time tests are conducted in a new application, where the input half-periods T(n) are initialized in advance without passing to measure them by the timer associated with their timed interrupts. All other processing of the SPLL algorithm remains functional, including the plots entrusted to Matlab via the RTDX link.

The technical characteristics of the constituent blocks of the DPLL simulated by Pspice must be attributed to the SPLL implemented on DSP so that it can behave like the DPLL. Thus, the VCO of the SPLL should exhibit a nonlinear characteristic similar to that of the VCO in figure 7. At least, the value of the sensitivity K_o must be drawn from the nonlinear characteristic according to the considered frequency. Thus, in the absence of directly measuring the filter voltages associated with the input test frequencies following the resolution of the graph of figure 7, we therefore measure on the characteristic of this figure, the two filter voltages x_1 and x_2 corresponding at the two adjacent frequencies y_1 and y_2 of each input frequency. From which, we draw by linear interpolation, the filter voltage U_f necessary to obtain the desired input frequency. The sensitivity K_o is deduced from this by calculation, table 1.

Input frequency	Y_1 (KHz)	X_1 (mV)	Y_2 (KHz)	X_2 (mV)	U_f (mV)	K_0 (KHz/V)	K_0 (KHz/V)
100 KHz	99.406	331.683	101.287	336.634	333.247	300.078	1885.10^3
200 KHz	199.708	599.010	201.525	603.960	599.806	333.441	2095.10^3
290.5 KHz	289.074	846.535	290.811	851.485	850.599	341.524	2146.10^3
625 KHz	623.986	1.9703 V	625.128	1.9753 V	1.9745 V	316.536	1989.10^3

Table. 1. Measurement of adjacent points relative to input frequencies and calculation of U_f and K_0

We will consider input half-periods T = 5 µs, 2.5 µs, 1.72 µs, and 0.8 µs, corresponding respectively to 100 kHz, 200 kHz, 290.5 kHz, and 625 kHz. The simulation time is 2 ms, like the one carried out with Pspice on the DPLL in figure 16. The optimal time constants of the loop filter established initially in Pspice, are to be used, τ_1=45µs and τ_2=11µs (R_1=450Ω, R_2=110Ω, C=0.1µF). They establish an optimal dynamic for the SPLL, through a damping factor of ε=0.7 and a natural frequency of 18.9 kHz ($\omega_n = 118.5 \times 10^3 \mathrm{rd/s}$) thus ensuring a sufficient bandwidth of the SPLL in the BF domain with minimal response time.

For the SPLL in its deferred version (not in real time), the voltages of the filter for the four input frequencies (100 KHz, 200 KHz, 290.5 KHz, 625 KHz) are gathered in figure 19. They all present a perfectly constant steady state, while the transient state extends less and less going from the proper response at the lowest test frequency, 100 kHz, to the highest corresponding to the frequency of 625 kHz.

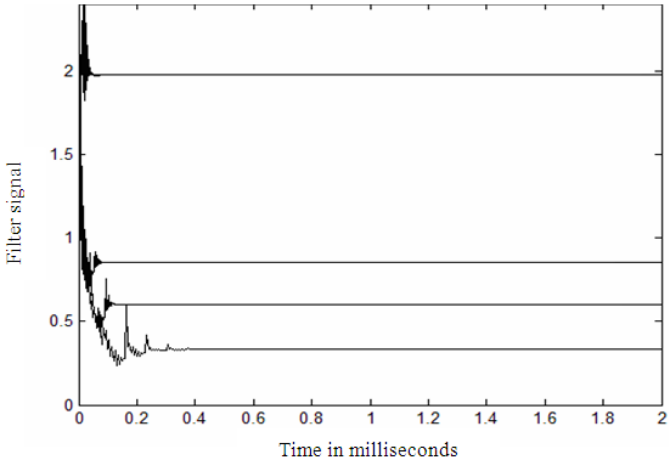


Fig. 19. Pooled filter voltages of the SPLL in response to the four test frequencies

IV.3 First comparison DPLL-SPLL

The Pspice simulation values referring to figure 16, were exported to data files for their graphical display using Matlab. Thus and for comparison, figure 20 groups together the four optimal filtered responses of the DPLL for the four test input frequencies.

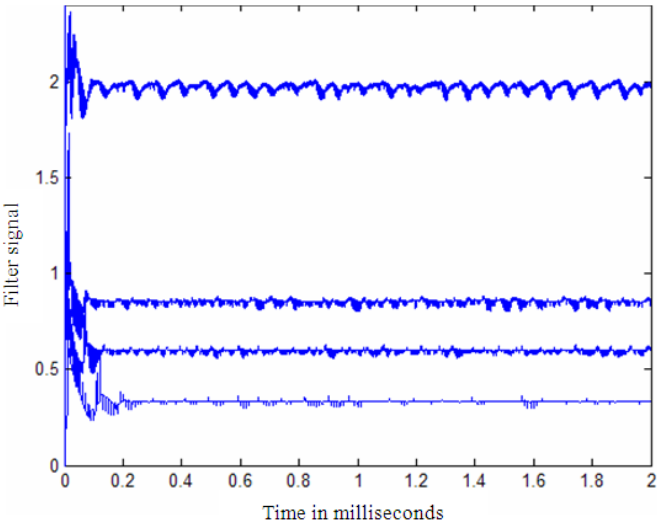


Fig. 20. Gathered filter voltages of the DPLL in response to the four test frequencies

Although the trends of the responses of the DPLL and the SPLL, are the same in the two figures 19 and 20, only, no oscillation of the SPLL persists in its steady state compared to the steady states provided by the DPLL. The transient states of the two figures 19 and 20 are more or less similar.

The figure 21 superimposes the filtered responses of the DPLL with those of the SPLL. The latter serve well as mean values in steady state, around which the former oscillate weakly.

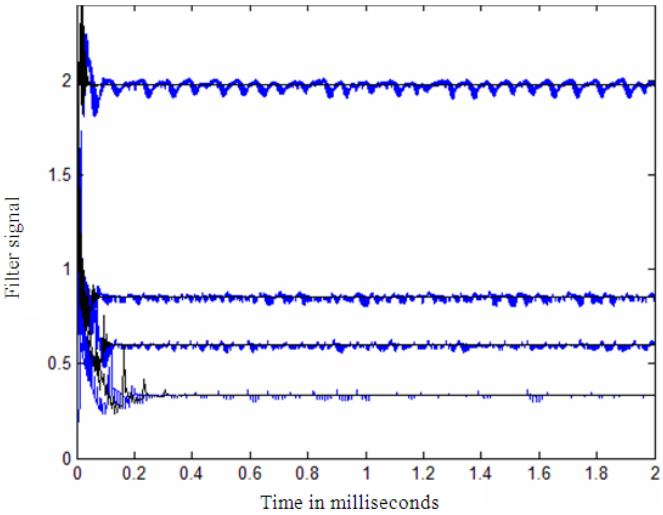


Fig. 21. Superimposed filter voltages of the DPLL and the SPLL in response to the four test frequencies

IV.4 Real-time acquisition of the input signal and measurement of filtered responses

To quantitatively evaluate the real-time execution of the SPLL application and its versions, we use the profiler, one of the tools of the integrated development environment, CCS.

IV.4.1 CCS profiling

CCS (3.1 version) allows the collection of execution statistics of certain regions of the code [5-13] by performing profiling which gives a measure of the performance of the application. Knowing that a machine cycle lasts 4.444 ns for the DSP C6713 under 225 MHz clock, the elapsed times expressed in terms of machine cycles consumed during the execution of the whole program or its procedures or even a number of lines of code are determined during profiling. This allows the verification of the execution in real time or to make optimizations in this direction.

IV.4.2 Back to real-time SPLL application

It contains two small hardware interrupt routines calling in common, a software interrupt routine for SPLL processing, figure 22. All the routines are included in the same C file grouping together the main program which initializes the processing.

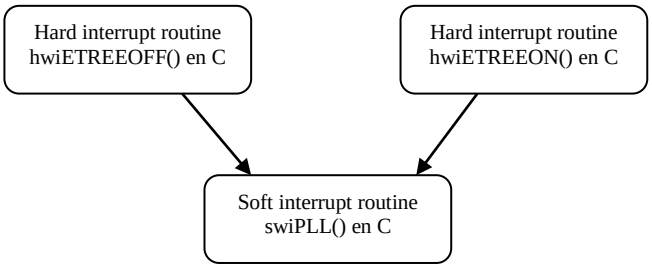


Fig. 22. General flowchart of hard and soft interrupt routines in C

The non-real-time SPLL version performed well for all input frequencies. But in real time, all the compile versions tried (optimized or not) did not allow real time execution even under 100 kHz input. Other precautions were taken in which, the memory was expanded to 0x400000 with a stack length of 0x1000. The program gets stuck in the while loop.

By reducing the soft routine to a minimum of SPLL processing, limiting itself to measuring the duration T of each state of the input signal, figure 23 shows that this measurement is then carried out in real time, but only for the frequency of 100 kHz (T = 5µs). The non-perfect constancy of the measurement is striking (maximum variation of ±0.4µs around the nominal value 5 µs). Must therefore consider further optimizing the program in its C version.

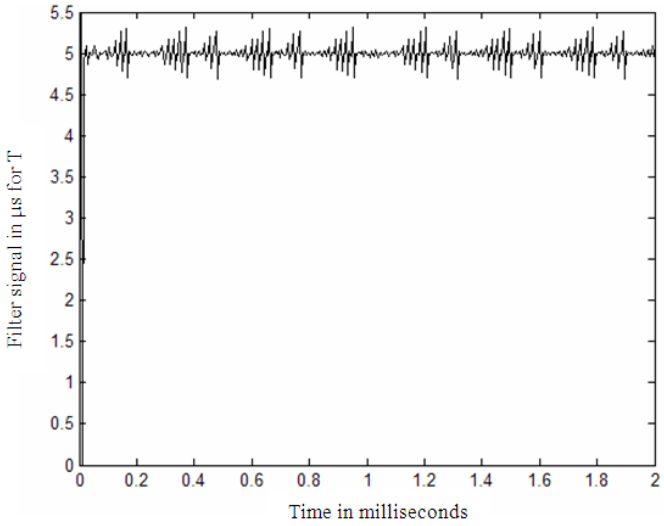


Fig. 23. Only real-time measurement of T proper at 100 kHz input

IV.4.3 Optimization of C code

One of the C optimization methods is the use of intrinsics [5-13]. These are special functions that are directly adapted to certain instructions of the C67x assembler set, thereby ensuring rapid optimization of C/C++ code. Our C code, however, has the particularity of not using any mathematical instructions to substitute, apart from the four usual arithmetic operations. But there again, no intrinsic instruction ensuring the addition, the subtraction, or the multiplication between reals with simple precision (type float), does not exist.

The many conditional (if else) statements held to be responsible for consuming time in each interrupt routine also do not have substitutable intrinsic statements. There remains the arithmetic operation of division which can be advantageously replaced by the reciprocal intrinsic function _rcpsp() of the denominator followed by a multiplication with the numerator, but this will require several instructions in order to arrive at a calculation precision on 32 bits [5-13]. The fact that each routine contains up to 5 division operations, this will require the preparation of a division procedure which will have to be called (with return) several times at each interrupt service, thereby making the operation more cumbersome. Nevertheless, this solution remains possible when one thinks of translating the two routines into linear assembler in which, the division procedure becomes a simple macro-instruction (without call-return).

In the first stage of optimization of the C code, the software routine of the SPLL processing is eliminated and its content is entrusted to each of the two hard routines in C. The SPLL processing provided by these

routines is not quite identical, it is depending on the state of the input signal specific to each routine.
The while statement has been tested outside of interrupts to confirm that it is never cycled through more than twice. It is therefore deployed twice to get rid of the branch instruction innate to this loop.
Only the "Program Level Opt. = None" and "Opt Level = None" (and not O2 or O3) was able to work under an input frequency of 100 kHz. We measure as profiling of the two interrupt routines hwiENTREE_OFF and hwiENTREE_ON: "total cycle" = 156452 + 162395 = 318847 cycles for 401 accesses, figure 24, that is 1417µs.

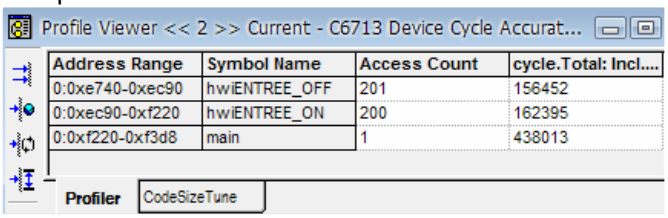


Fig. 24. Profiling SPLL processing hard interrupt routines

However, in 2 ms of simulation, 400 logic states of the input signal appear (at the frequency of 100 kHz). Each state lasts 5 µs and therefore consumes 1417/401 = 3.53 µs of processing. This processing is therefore carried out in real time but under 100 kHz and not under 200 kHz with T = 2.5 µs. Figures 25 and 26 confirm this aspect at 100 kHz. The slight widening of the line at 5 µs in figure 25 is probably due to the difference in processing length following the various conditional branches of the code. This effect has a weak effect on the filter response in figure 26, where dots appear on the steady state line.

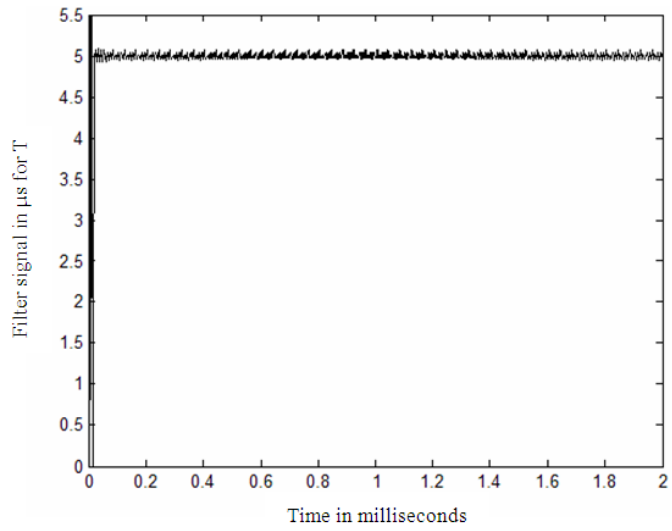


Fig. 25. Real-time measurement of the duration of each state of the input signal at 100 kHz

From the profiler measurements, we see that a thorough optimization of the code is necessary to serve in real time, higher input frequencies. This requires a rewriting in assembly language of the many tests and branches of the SPLL program in order to increase its speed of execution.
The target language is linear assembler, since it is a macro-assembler located halfway between an advanced language such as C/C++ and pure assembly language. It therefore achieves a good compromise between simplicity of writing (therefore readability or portability) and speed of execution.

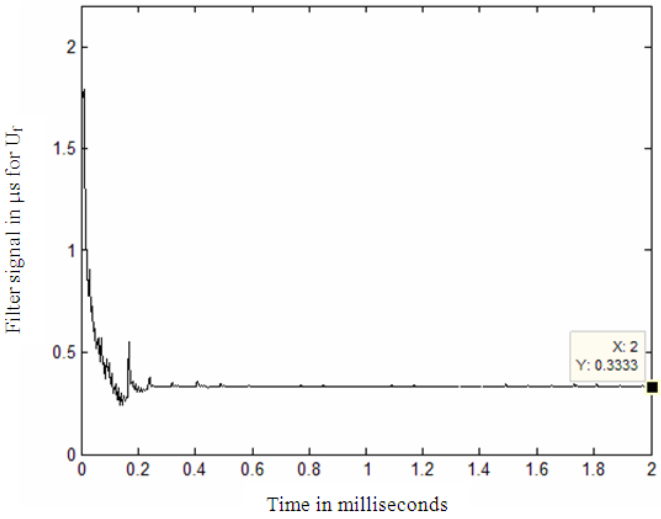


Fig. 26. Response of the loop filter under 100 kHz input in accordance with that of Pspice

IV.4.4 Optimization by linear assembly

The two hard routines are again rudimentary and reside in two separate C files. They call in common a single SPLL processing procedure written in linear assembly, passing it as arguments the content of an interrupt counter as well as the current state of the input signal, figure 27. The procedure after SPLL processing, increments the interrupt counter and communicates its new content to the calling routine.

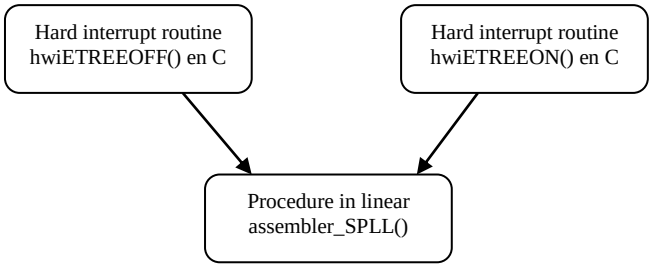


Fig. 27. General flowchart of hard interrupt routines calling a procedure in linear assembly language

In order to interface a procedure in assembler to the C program, it is necessary to follow the register conventions of the C/C++ environment [5-13].

For an input frequency of 200 kHz, we measure as profiling the two interrupt routines hwiENTREE_OFF and hwiENTREE_ON, as well as the SPLL procedure: "total cycle:" = 108173 + 130316 + 212033 = 450522 cycles for 803 accesses, figure 28, that is 2002µs.

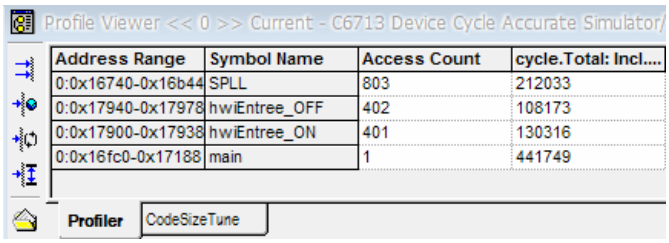


Fig. 28. Profiling of hard interrupt routines and SPLL processing procedure in linear assembly

For the 803 logic states of the input signal at the frequency of 200 kHz, each state lasts 2.5 µs and therefore consumes 2002 / 803 = 2.49 µs of processing. This processing is done just in real time under 200 kHz and therefore not under 290 kHz with T = 1.73 µs. Figures 29 and 30 confirm this aspect at 200 kHz.

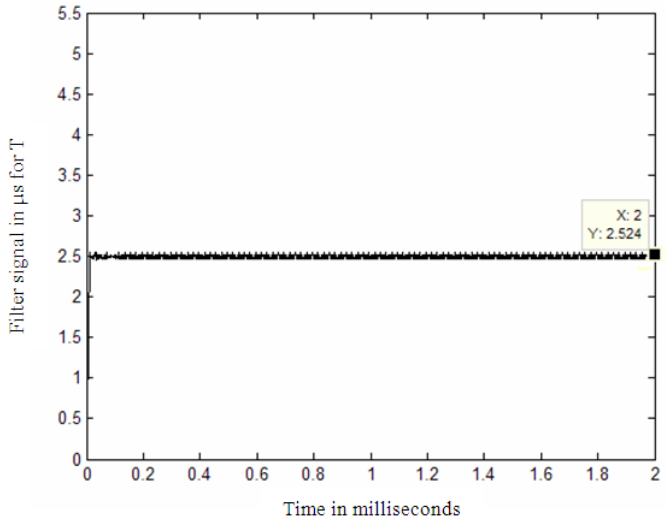


Fig. 29. Real-time measurement of the duration of each state of the input signal at 200 kHz

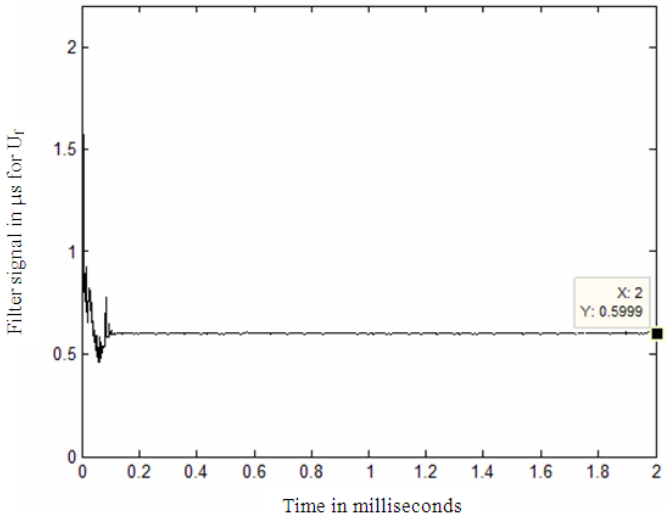


Fig. 30. Response of the loop filter under 200 kHz input in accordance with that of Pspice

At 290 kHz, the application no longer becomes real-time. We thus move on to the last solution provided by the assembly language by optimizing it manually.

IV.4.5 Optimization of the assembler version

The proposed Method for the parallelism of the assembly code (manual method): The parallelism of the code amounts to the paralleling of independent instructions and processed by different units. Thus, in addition to the recommendations [5-13] relating to the maximum diversification of the functional units to be used and the minimum use of their crossed paths, it is necessary:

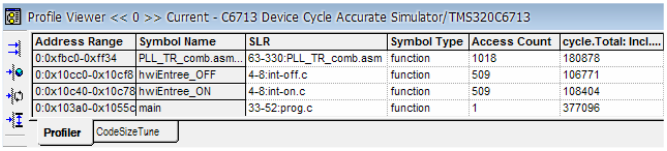
a- List by code areas, the instructions independent of each other that can be executed in parallel.

b- Parallelize the maximum number of instructions per code zone by diversifying the functional units used and by doubling if necessary certain variables according to the two rows of registers, A and B, with a view to more efficient use in the rest of the coded.

c- Minimize in some cases the no-operations (NOPs) related to multicycle instructions by replacing them with code-independent instructions.

As in linear assembly, the two interrupt routines in C call on a single SPLL processing procedure written manually in pure (non-linear) assembly language. This procedure is optimized in the parallelization of its instructions in order to take full advantage of the availability of the DSP pipeline, thus allowing to obtain an increased speed of execution.

Thus, for an input frequency of 290 kHz (input states of duration 1.724 µs), we find as profiling: "total cycle" = 180878 + 106771 + 108404 = 396053 cycles. That is 1760 µs in 1018 accesses, figure 31. It was found that the program cannot exceed 1018 measurement points, while it takes 1156 points to reach the 2 ms simulation time.


Profile Viewer << 0 >> Current - C6713 Device Cycle Accurate Simulator/TMS320C6713

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.Total: Incl....
0:0xfbc0-0xff34	PLL_TR_comb.asm...	63-330:PLL_TR_comb.asm	function	1018	180878
0:0x10cc0-0x10cf8	hwiEntree_OFF	4-8:int-off.c	function	509	106771
0:0x10c40-0x10c78	hwiEntree_ON	4-8:int-on.c	function	509	108404
0:0x103a0-0x1055c	main	33-52:prog.c	function	1	377096

Profiler CodeSizeTune

Fig. 31. Profiler statistics window under 290kHz input

From which we obtain 1.724 µs ≈ (1760 / 1018 = 1.728 µs), which estimates that the execution can most likely be in real time for an input frequency of 290 kHz, following the very close values found as in the case of 200 kHz. Figures 32 and 33 confirm the real-time execution aspect, whose steady-state plot is graphically extended up to the instant 2 ms. But an execution under an input frequency of 625 kHz with a state duration of 0.8 µs cannot be carried out in real time.

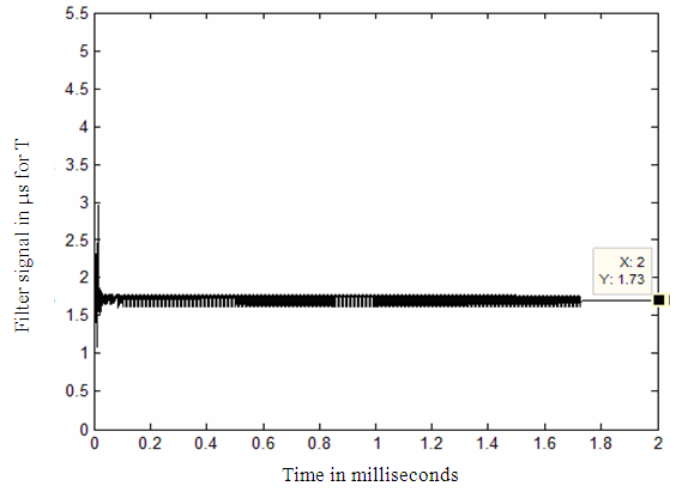


Fig. 32. Real-time measurement of the duration of each state of the input signal at 290 kHz

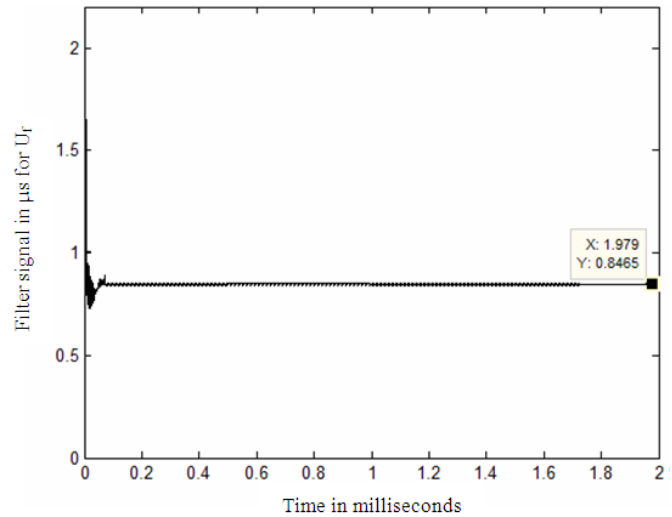


Fig. 33. Response of the loop filter under 290kHz input in accordance with that of Pspice

From figures 28 and 31, the profiling indicates a time consumed in number of machine cycles of the two combined hardware interrupt routines, even when reduced to a minimum, which remains equivalent to that consumed by the SPLL processing procedure.

IV.4.6 Second comparison DPLL-SPLL

The figure 34 confirms the aspect of execution in real time under the first three test frequencies, easily covering the 1156 measurement points for the frequency of 290 kHz corresponding to a time of 2 ms. The plots obtained are comparable to those resulting from SPLL in deferred time of the C language, figure 35, themselves in conformity with the results of the Pspice simulation, figures 20 and 21.

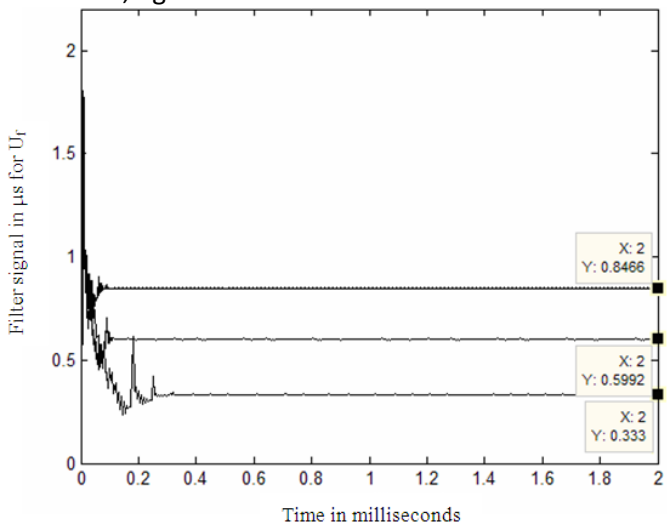


Fig. 34. Real-time assembler results under 100, 200, and 290 kHz input frequencies

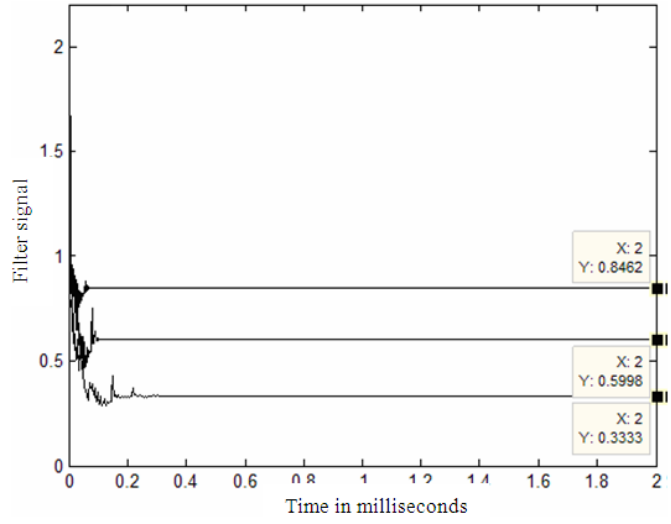


Fig. 35. Results of C in delayed time under input frequencies 100, 200, and 290 kHz

The two hard interrupt routines in C are now fully converted to assembly language. To do this, saving and restoring the context of the registers used in a stack are carried out manually by the two routines [5-13]. Thus, for an input frequency of 290 kHz (input states lasting 1.73nµs), we find as profiling:

"total cycle" = 109078 + 107749 = 216827 cycles. That is 963.7 µs in 1156 accesses, figure 36. Hence 1.724 µs > (963.7 / 1156 = 0.83 µs) implying an easier execution of the code in real time than that of the previous case.

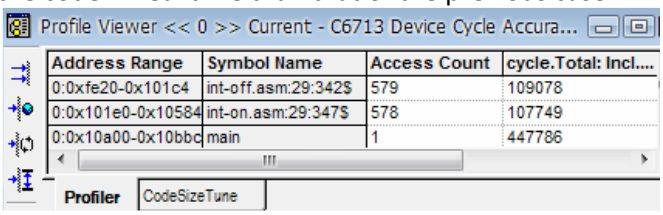


Fig. 36. Profiling hard SPLL processing interrupt routines in assembler

Operation under an input frequency of 625 kHz still remains inaccessible to real-time execution, because the profiling in figure 36 denounces and highlights, a maximum real-time execution frequency that cannot exceed $\frac{1}{2\times 0.83\mu s} = 600$ KHz.

V. CONCLUSION

A design assisted by Pspice was carried out in order to achieve an optimized DPLL in its static and dynamic behavior. The second-order transmittance obtained from the closed-loop DPLL was assigned an optimal damping factor of $1/\sqrt{2}$, thus reducing the response time of the DPLL to its minimum. The implementation of the SPLL algorithm on a 32-bit floating point DSP was carried out using a C6713 simulator from Texas Instruments. Most of the application is shared between the acquisition of the input signal using two external hardware interrupt routines, the SPLL processing provided by a software interrupt routine, and the graphical representation of the results on Matlab based on an RTDX binding. In the first version of the C program, the real-time execution of the SPLL faded for all input test frequencies from 100 kHz. The steady states obtained from the SPLL are all free of residual oscillations and are reached in a response time not exceeding 0.2 ms. These optimal responses constitute average values for the damped oscillatory responses of the DPLL. Returning to the real-time execution aspect, optimization of the C code allowed real-time SPLL processing under only 100 kHz input. The translation of SPLL processing into linear assembly language has made it possible to widen the input field processed at the double frequency of 200 kHz. The use of pure and manually optimized assembler made it possible to reach the frequency of 290 kHz by using two rudimentary hardware interrupt routines in C language calling in common, the SPLL procedure in assembler. Sharing SPLL processing between two hardware interrupt routines written exclusively in assembler has doubled the limits of real-time processing. That is to say, around 600 kHz.

REFERENCES

[1] R. E. Best, « Phase-Locked Loops: Design, Simulation and Applications », 4th Edition, McGraw-Hill, 1999.
[2] Wikipedia, the free encyclopedia, « Phase-locked loop » at http://en.wikipedia.org/wiki/Phaselocked_loop.
[3] J. Encinas, « Phase Locked Loop system (PLL): achievements and applications », Masson edition, 1990.
[4] F. M. Gardner, « Phaselock Techniques », 2nd Edition, John Wiley and Sons, 1979.
[5] Texas Instruments, « TMS320C6000 Instruction Set Simulator », Technical Reference, spru609f, 2005.
[6] Texas Instruments, « Code Composer User's Guide », spru296.pdf, 1999.
[7] Texas Instruments, « TMS320C6000 Programmer's Guide », spru198g.pdf, 2002.
[8] Texas Instruments, « TMS320C6000 Optimizing Compiler User's Guide », spru187i.pdf, 2001.
[9] Texas Instruments, « TMS320C6000 CPU and Instruction Set Reference Guide », spru189f.pdf, 2000.
[10] Texas Instruments, « TMS320C6000 DSP/BIOS Application Programming Interface Reference Guide », spru403c.pdf, 2001.
[11] Texas Instruments, « TMS320C6000 DSP/BIOS User's Guide », spru423a.pdf, 2001.
[12] Texas Instruments, « TMS320C6000 Chip Support Library API User's Guide », spru401c.pdf, 2001.
[13] Texas Instruments, « TMS320C6000 Peripherals Reference Guide », spru190d.pdf, 2001.
[14] P. Tobin, « Pspice for digital communication engineering », Morgan & Claypool publishers series #10, 2007.
[15] P. Tobin, « Pspice for analog communications engineering », Morgan & Claypool publishers series #9, 2007.
[16] Orcad, « Orcad Pspice A/D », Reference Manual, version 9, 1998.