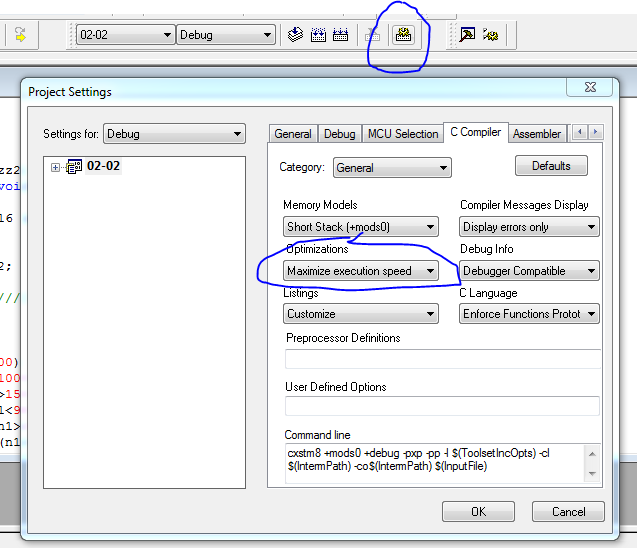
Настроим оптимизацию



----------------==================

/////////////////

u8 NT1 = 100;

///////////////////////////////////////////////////////

/// классичечкий двоичный поиск

u8 b\_f(u16 A) // A - что ищем по таблице T1 длиной NT1

{

u8 m, left, right;

left = (u8)0; right = (u8)(NT1-1);

while (1)

{

if (left > right) return (0xff); // значение не найдено

m = (u8)(left + (right - left) / 2);

if (T1[m] < A) left = (u8)(m + 1);

if (T1[m] > A) right = (u8)(m - 1);

if (T1[m] == A) return m;

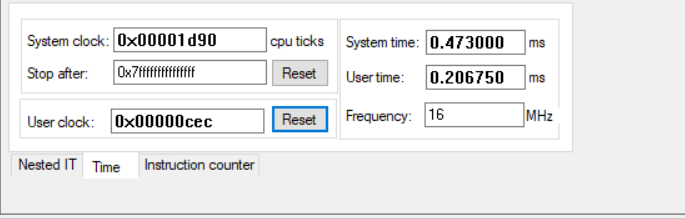
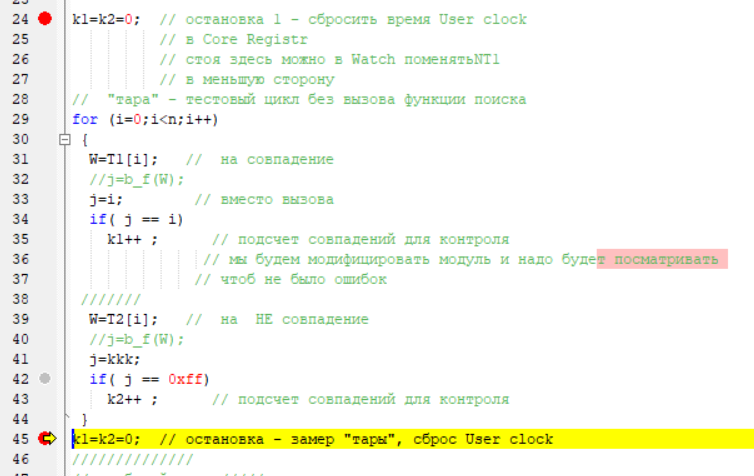
}

}

Это базовая функция, замеряем среднюю скорость по тесту и распишем как мы это делаем:

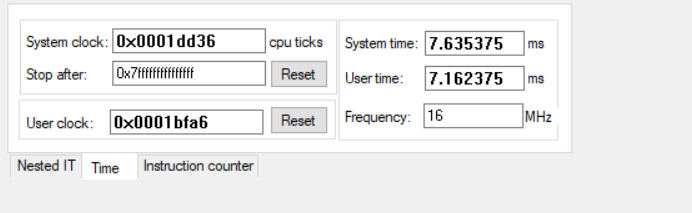
Шаг 1:

Замеряем тестовый цикл:



Шаг 2:

Замеряем рабочий цикл



Отнимаем ‘тару’ = 7.162375-0.206750 = 6,955625

Далее последовательно делаем изменения в надежде получить более быструю скорость-длительность выполнения функции , сравнивая с предыдущим замером. Переименовываем функцию, чтоб сохранить наши шаги. Шаг-попытка может быть неудачной, возвращаем изменение и делаем другую попытку.

1. Сначала , очевидно, уменьшим число ифов

u8 b\_f1 (u16 A) // 1

{

u8 m, left, right;

left = (u8)0; right = (u8)(NT1-1);

while (1)

{

if (left > right) return (0xff); // значение не найдено

m = (u8)(left + (right - left) / 2);

if (T1[m] == A) return m;

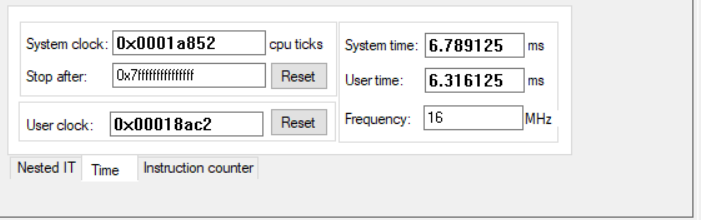
if (T1[m] < A) left = (u8)(m + 1);

else right = (u8)(m - 1);

}

}

Итог: заметно быстрее)



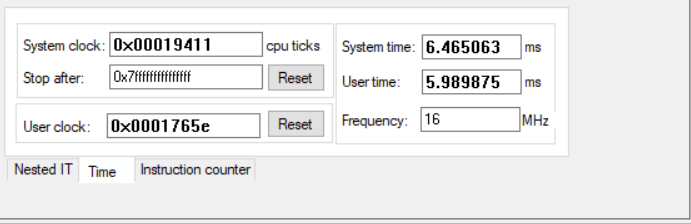
Отнимаем ‘тару’ = 6.316125-0.206750 = 6,109375

**1a)** (Может такое пойдёт) if (T1[m] == A) return m; закомментил и засунул вот так

if (T1[m] < A) left = (u8)(m + 1);

else if (T1[m] == A) return m;

else right = (u8)(m - 1);

Итог: 

Отнимаем ‘тару’ = 5.989875-0.206750 = 5,783125

1. Следующая версия – уменьшение обращение к индексированным переменным

u8 b\_f2 (u16 A) // 2

{

u8 m, left, right;

u16 r;

left = (u8)0; right = (u8)(NT1-1);

while (1)

{

if (left > right) return (0xff); // значение не найдено

m = (u8)(left + (right - left) / 2);

r=T1[m];

if (r == A) return m;

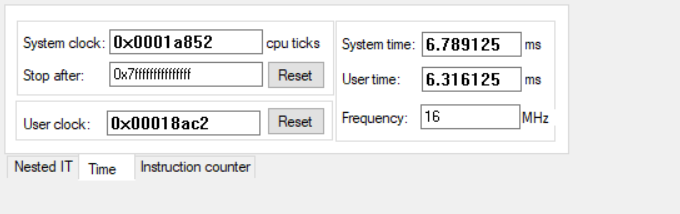
if (r < A) left = (u8)(m + 1);

else right = (u8)(m - 1);

}

}

Итог: Без изменений, идентичный результат пункта 1.



Отнимаем ‘тару’ = 6.316125-0.206750 = 6,109375

1. Иногда с глобальными переменными программа работает быстрее - проверим

u8 m, left, right;

u16 r;

u8 b\_f3 (u16 A) // 3

{

//u8 m, left, right;

//u16 r;

left = (u8)0; right = (u8)(NT1-1);

while (1)

{

if (left > right) return (0xff); // значение не найдено

m = (u8)(left + (right - left) / 2);

r=T1[m];

if (r == A) return m;

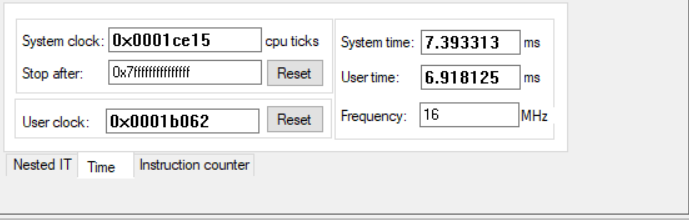
if (r < A) left = (u8)(m + 1);

else right = (u8)(m - 1);

}

}

Итог: время работы увеличилось, можно сделать вывод, что самая медленная реализация является со множеством if и рассматриваемая



Отнимаем ‘тару’ = 6.918125-0.206750 = 6,711375

В рамках легкой модификации алгоритма – больше я не могу придумать, что бы еще ускорило алгоритм.

Чтобы объяснить появление следующей гипотезы модификации алгоритма в сторону ускорения, хочется сравнить двоичный поиск и последовательный.

Напиши функцию простейшего последовательного поиска

u8 P\_f (u16 A) //

{

u8 m;

for(m=0;m<NT1;m++)

{

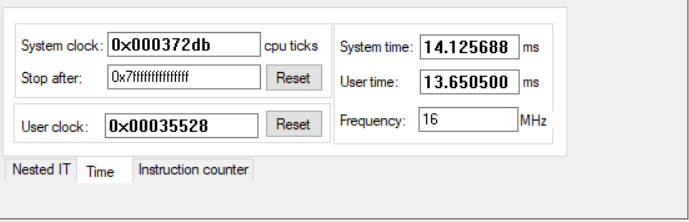
if (T1[m] == A) return m;

}

return (0xff);

}

Результат: Выходит, что последовательный в 2 раза дольше отрабатывает, чем двоичный при NT=100

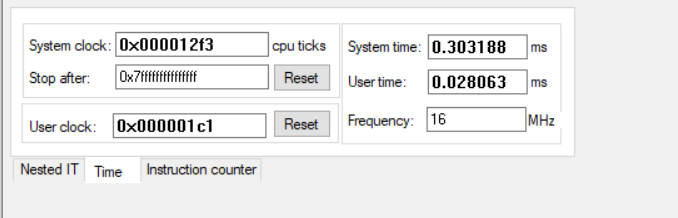


И давай поизмеряем оба и сравним – построим график – когда двоичный поиск становится быстрее последовательного?

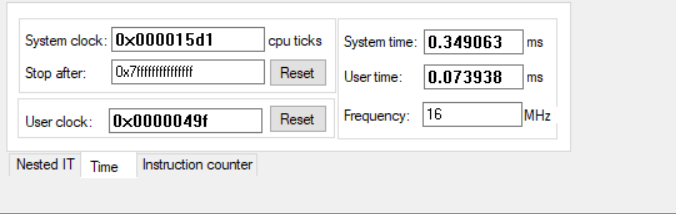
Надо замерить обе функции для NT1 = 3,4,5,6,7,8,9,10 - думаю , на этом участке мы увидим расхождение по скорости в пользу двоичного поиска. Но, мне кажется, в начале, при NT1 = 3 – последовательный будет быстрее. Значит где-то на этом участке графики пересекутся , обозначив участок, где последовательный быстрее.

NT1 = 3:

Последовательный:

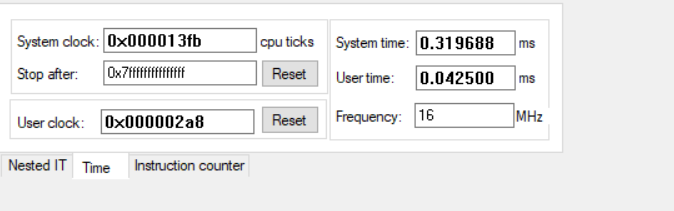


Двоичный:

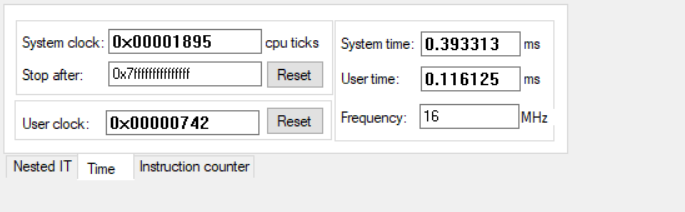


NT1 = 4:

Последовательный:

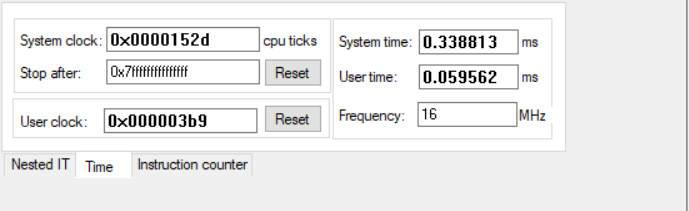


Двоичный:

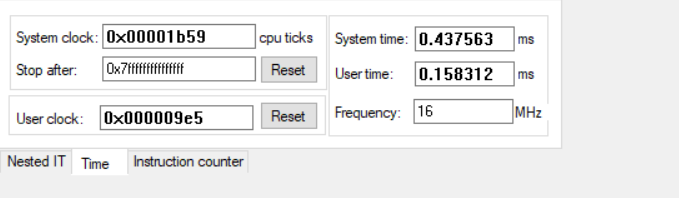


NT1 = 5:

Последовательный:

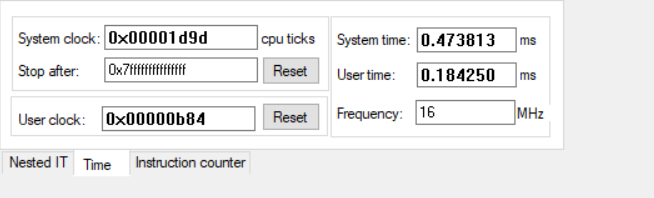


Двоичный:

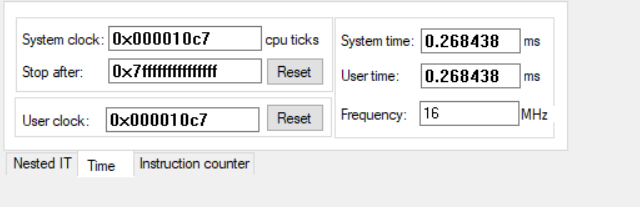


NT1 = 10:

Последовательный:

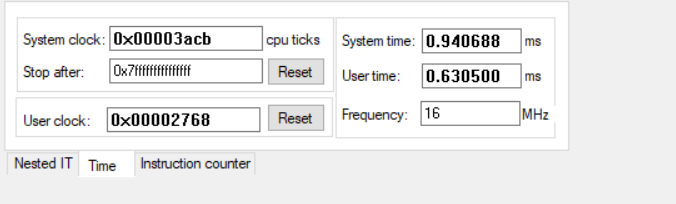


Двоичный:

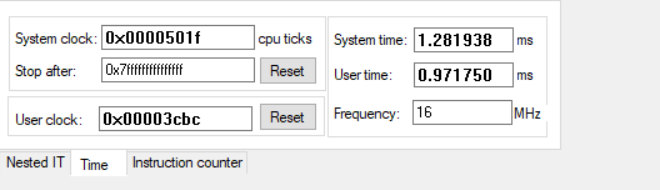


NT1 = 20:

Последовательный:

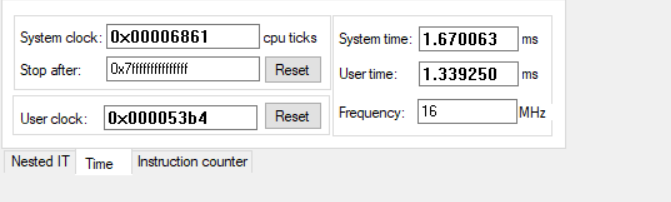


Двоичный:

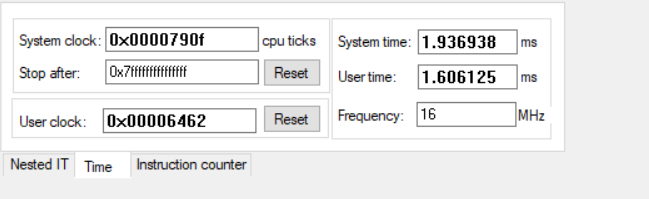


NT1 = 30:

Последовательный:

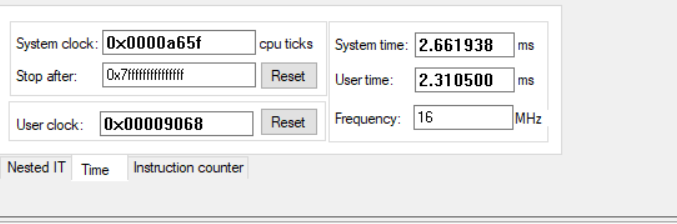


Двоичный:

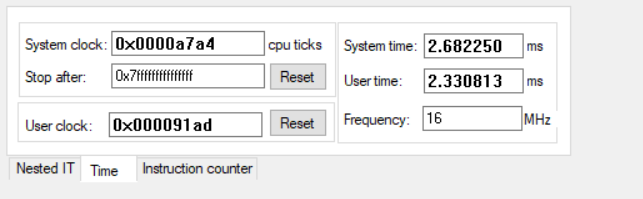


NT1 = 40:

Последовательный:

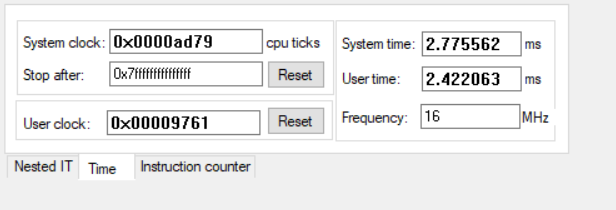


Двоичный:

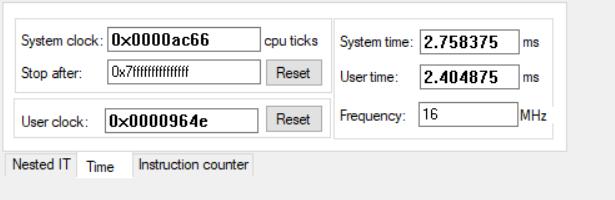


**NT1 = 41**(нашёл!! Вот тут двоичный начинает работать быстрее):

Последовательный:

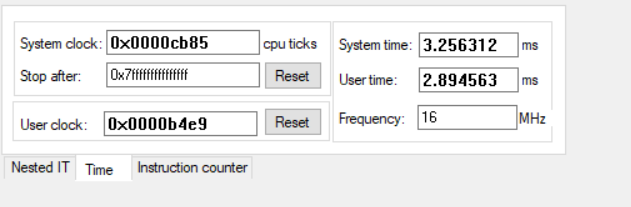


Двоичный:

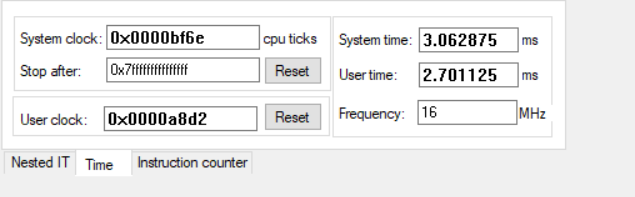


NT1 = 45:

Последовательный:

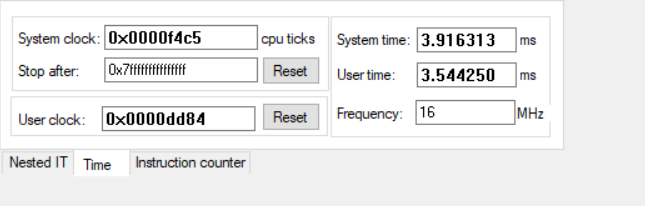


Двоичный:

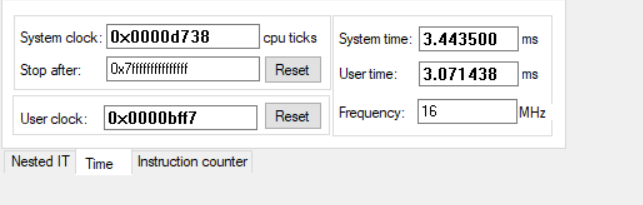


NT1 = 50:

Последовательный:



Двоичный:



Этот факт приводит к желанию посмотреть, как себя ведет гибридный алгоритм

Для того же NT1=100, поиграть NNT.

U8 NNT=3;// или больше?? Это размер последнего куска деления

// когда переходить на

// последовательный поиск

u8 bp\_f4 (u16 A) // гибрид бинарного и последовательного

{

u8 m, left, right;

u16 r;

left = (u8)0; right = (u8)(NT1-1);

while (1)

{

// if (left > right) return (0xff);

// вместо деления до победного конца – остановится раньше

// и перейти к последовательному поиску

if ((right - left) < NNT)

{

for(m= left ;m<= right;m++)

{

if (T1[m] == A) return m;

}

return (0xff);

}

}// конец посл поиска и программы

//

m = (u8)(left + (right - left) / 2);

r=T1[m];

if (r == A) return m;

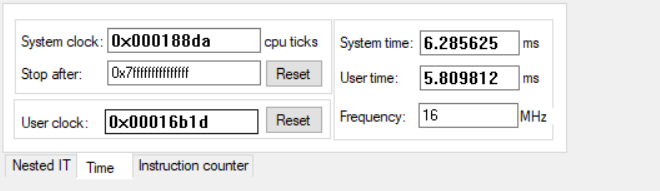
if (r < A) left = (u8)(m + 1);

else right = (u8)(m - 1);

}

}

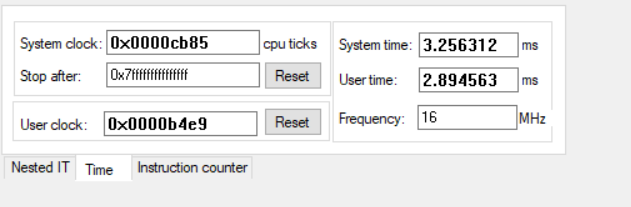
Результат: Гибрид быстрее при NT1=100, чем двоичный и последовательный. NNT потестил с 10,20 – итог, быстрее отрабатывает ~5.3 ms user time.



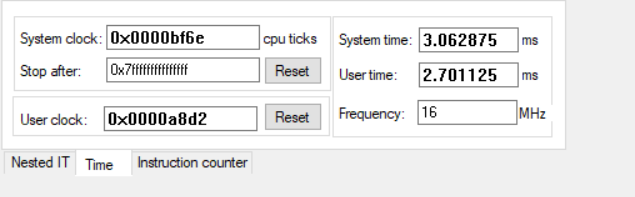
**Общий итог**: Последовательный побеждает двоичный до NT=41 не включая и выигрывает гибридный до NT= ~30. Но при NT=100 гибридный выигрывает последовательный и двоичный, даже несмотря на то, что мы как-то оптимизируем двоичный, он всё равно проигрывает.

NT1 = 41:

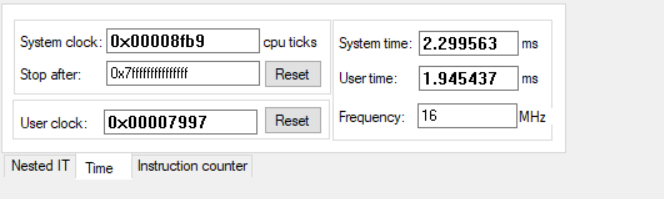
Последовательный:



Двоичный:

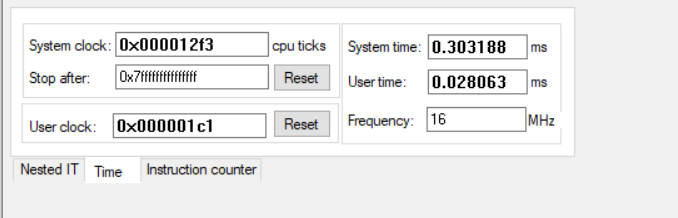


Гибрид:

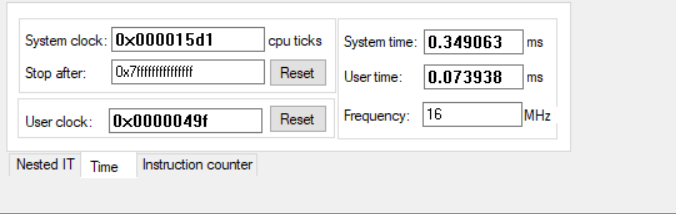


NT1 = 3:

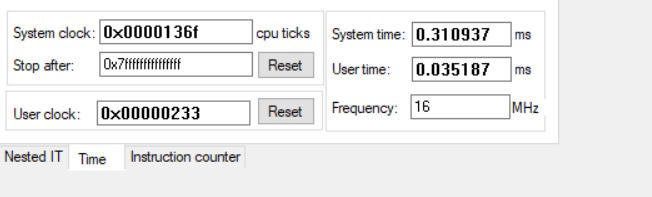
Последовательный:



Двоичный:



Гибрид:



Все это - акт №1 – определить наиболее быстрый алгоритм, для конкретного процессора и компилятора.

И тогда перейдем к Акту №2

///////////////////