HT-IDE3000 Programmer's Guide

for Holtek C Language

Ver 1.1

NOTICE

Contents

Chapter 1
Holtek C Language

1

Introduction

The Holtek C compiler is based on ANSI C. Due to the architecture of the Holtek micro-controller, only a subset of ANSI C is supported. This chapter describes the C programming language supported by the Holtek C compiler. They will apply to both types of compilers if there is no special mention.

This chapter covers the following topics:

- C program structure
- Identifiers
- Data types
- Constants
- Operators
- Program control flow
- Functions
- Pointers and arrays
- Structures and unions
- Preprocessor directives
- Language extensions and restrictions

C Program Structure

A C program is a collection of statements, comments, and preprocessor directives.

Statements

Statements, which may consist of variables, constants, operators and functions, are terminated with a semicolon and perform the following operations:

- Declare data variables and data structures
- Define data space
- Perform arithmetic and logical operations
- Perform program control operations

One line can contain more than one statement. Compound statements are one or more statements contained within a pair of braces and can be used as a single statement. Some statements and preprocessor directives are required in the Holtek C source files. The following is a shell:

```
void main()
{
/* user application source code */
}
```

The *main* function is defined within the user application source code. There may be more than one source file for an application, but only one source file can contain the *main* function.

Comments

Comments are used to document the meaning and operation of the source statements and can be placed anywhere in a program except for the middle of a C keyword, function name or variable name. The C compiler ignores all comments. Comments cannot be nested. The Holtek C compiler supports two kinds of comments, block comment and line comment.

➔ Block comment

The block comment begins with **/*** and ends with ***/**, for example:

```
/* this is a block comment */
```

A block comment's end character ***/** may be placed in a different line from the beginning block comment characters. In this case all the characters between the starting comment characters and end comment characters, are treated as comments and ignored by the C compiler.

➔ Line comment

A line comment begins with // and comments out all characters to the end of the line, for example

```
// this is a line comment
```

Identifiers

The name of an identifier contains a sequence of letters, digits, and under scores with the following rules:

- The first character must not be a digit.

- Only the first 31 characters are significant.
- Upper case and lower case letters are different.
- Reserved words cannot be used.

Reserved words

The following are the reserved words supported by the Holtek C complier. They must be in lower case.

auto	bit	break	case	char
const	continue	default	do	else
enum	extern	for	goto	if
int	long	return	short	signed
static	struct	switch	typedef	union
unsigned	void	volatile	while	

The reserved words **double**, **float** and **register** are not supported by the Holtek C compiler.

Data types

Data types and sizes

Four basic data types are supported by the Holtek C compiler,

```
bit     a single bit
char    a single byte holding one character
int     an integer occupying one byte
void    an empty set of values, used as the type returned
        by functions that generate no value
```

The following qualifiers are allowed

Qualifier	Applicable Data Type	Use
const	any	place the data in a ROM space
long	int	create a 16-bit integer
short	int	create an 8-bit integer
signed	char, int	create a signed variable
unsigned	char, int	create an unsigned variable

The following are the data types, sizes and range

Data Type	**Size (bits)**	**Range**
bit	1	0,1
char	8	-128 ~ 127
unsigned char	8	0 ~ 255
short int	8	-128 ~ 127
unsigned short int	8	0 ~ 255
int	8	-128 ~ 127
unsigned	8	0 ~ 255
long	16	-32768 ~ 32767
unsigned long	16	0 ~ 65535

Declaration

Variables must be declared before being used as this defines the data type and the size of the variable. The syntax of variable declaration is:

data_type variable_name [, *variable_name* . . .] ;

where *data_type* is a valid data type and *variable_name* is the name of the variable. The variables declared in a function are private (or local) to that function and other functions cannot access these variables directly. The local variables in a function exist and are valid only when this function is called, and are non-valid when exiting from the function. If the variable is declared outside of all functions, then it is global to all functions.

The qualifier **const** can be applied to a declaration of any variable, to specify that the value of the variable will not be changed. The variables declared with **const** are placed within the ROM space. The **const** qualifier can be used in array variables. A **const** variable must be initialized upon declaration, followed by an equal sign and an expression. Other variables cannot be initialized when declared.

A variable can be declared in a specified RAM address by using the @ character; the syntax is:

data_type variable_name **@** *memory_location ;*

The *memory_location* specifies the address variable located. To allocate a variable above the RAM bank 0 in the multiple RAM banks MCU, you might specify the bank no. in the high byte of *memory_location*. You should check the data sheet of the Holtek MCUs to get the information of the available RAM space.

For example:

```
int v1 @ 0x40; // declare v1 in the RAM bank 0 offset 0x40
int v2 @ 0x160;// declare v2 in the RAM bank 1 offset 0x60
```

Also, an array can be declared in a specified location:

```
int port[8] @ 0x20; // array port takes memory location
                    // 0x20 through 0x27
```

All variables implemented by the Holtek C compiler are static unless they are declared as external variables. Note that both static and external variables will not be initialized to zero by default.

Note: Declaring a variable as unsigned type will get more efficient code than as signed.

Constants

A constant is any literal number, single character or character string.

Integer constants

An integer constant is evaluated as int type, a long constant is terminated with l or L. Unsigned constants are terminated with a u or U, the suffix ul or UL indicates unsigned long. The value of an integer constant can be specified with the following forms:

Binary constant: preceding the number by 0b or 0B
Octal constant: preceding the number by 0 (zero)
Hexadecimal constant: preceding the number by 0x or 0X
Others not included above are decimal

Character constants

A character constant is an integer, which is denoted by a single character enclosed by single quotes. The value of a character constant is the numeric value of the character in the machine s character set. ANSI C escape sequences are treated as a single character constant.

Escape Character	**Description**	**Hex Value**
\a	alert (bell) character	07
\b	backspace character	08
\f	form feed character	0C
\n	new line character	0A
\r	carriage return character	0D
\t	horizontal tab character	09
\v	vertical tab character	0B
\\	backslash	5C
\?	question mark character	3F
\'	single quote (apostrophe)	27
\"	double quote character	22

String constants

String constants are represented by zero or more characters (including the ANSI C escape sequences) enclosed in double quotes. A string constant is an array of characters and has an implied null (zero) value after the last character. Hence, the total required storage is one more than the number of the characters within the double quotes.

Enumeration constants

Another method for naming integer constants is called enumeration. For example:

```
enum {PORTA, PORTB, PORTC} ;
```

defines three integer constants called enumerators and assigns values to them.

The enumeration constants have type **int** (-128~127). An explicit integer value might be associated with an enumberation constants. For example,

```
enum {BIG=10, SMALL=20} ;
```

The first enumeration constant has the value 0 if no explicit value is specified. Subsequent enumeration constants without explicit associations receive an integer value one greater than the value associated with the previous enumeration constant.

An enumeration can be named. For example:

```
enum boolean {NO, YES};
```

The first name (NO) in an **enum** statement has the value 0, the next has the value 1.

Operators

An expression is a sequence of operators and operands that specifies a computation. An expression follows the rules of algebra, may result in a value and may cause side effects. The order of evaluation of subexpressions is determined by the precedence and grouping of the operators. The usual mathematical rules for associativity and commutativity of operators may be applied only where the operators are really associative and commutative. The different types of operators are discussed in the following.

Arithmetic operators

There are five arithmetic operators,

+	addition
-	subtraction
*	multiplication
/	division
%	modulus (the remainder of division, always positive or zero)

The modulus operator %, can only be used with integral data types.

Relational operators

The relational operators compare two values and return either a TRUE or FALSE result based on the comparison.

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Equality operators

The equality operators are exactly analogous to the relational operators

= =	equal to
!=	not equal to

Logical operators

The logical operators support the logical operations AND, OR and NOT. They create a TRUE or FALSE value. Expressions connected by && and || are evaluated from left to right. The evaluation stops as soon as the result is known. The numeric value of a relational or logical expression is 1 if the relation is true, and 0 otherwise. The unary negation operator ! converts a non-zero operand into 0 and a zero operand into 1.

&&	logical AND
||	logical OR
!	logical NOT

Bitwise operators

There are six operators for manipulating bit-by-bit operations. The shift operators >> and << perform the right and left shifts of the left operand by the number of bit positions given by the right operand, which must be positive. The unary ~ yields the one's complement of an integer, converts every 1-bit to a 0-bit and vice versa.

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	one's complement
>>	right shift
<<	left shift

Assignment operators

There are a total of 10 assignment operators for expression statements. For simple assignment, the equal sign is used with the value of the expression replacing the variable, in the left operand. This also provides a shortcut for modifying a variable by performing an operation on itself.

```
<var> + = <expr>  add the value of <expr> to <var>
<var> - = < expr> subtract the value of <expr> from <var>
<var> * = <expr>  multiply <var> by the value of <expr>
<var> / = <expr>  divide <var> by the value of <expr>
<var> % = <expr>  modulus, remainder when<var>is divided
                  by <expr>
<var> & = <expr>  bitwise AND <var> with the value of <expr>
<var> | = <expr>  bitwise OR <var> with the value of <expr>
<var> ^ = <expr>  bitwise XOR <var> with the value of <expr>
<var> >> = <expr> right shift <var> by <expr> positions
<var> << = <expr> left shift <var> by <expr> positions
```

Increment and decrement operators

The increment and decrement operators can be used in a statement by themselves, or can be embedded within a statement with other operators. The position of the operator indicates whether the increment or decrement is to be performed before (prefix operators) or after (postfix operators) the evaluation of the statement it is embedded within.

```
++ <var>       pre-increment
<var> ++       post-increment
```

```
- -<var>      pre-decrement
<var>- -      post-decrement
```

Conditional operators

The conditional operator **?:** is a shortcut for executing a statement between two selectable statements according to the result of the expression.

```
<expr> ? <statement1> : <statement2>
```

If <*expr*> evaluates to a nonzero value, <*statement*1> is executed. Otherwise, <*satement*2> is executed.

Comma operator

A pair of expressions separated by a comma is evaluated from left-to-right and the value of the left expression is discarded. All side effects of the left expression are performed before the evaluation of the right expression. The type and value of the result are the type and value of the right operand. For example,

f (a, (t=3, t+2), c) ;

has three arguments, the second of which has the value 5.

Precedence and associativity of operators

The following table lists the precedence and associativity of operators. The precedence is from the highest to the lowest. Each box holds operators with the same precedence. Unary and assignment operators are right associative, all others are left associative.

Operators	**Description**	**Associativity**
[]	subscription	left to right
()	parenthesis	
->	structure pointer	
.	structure member	
sizeof	size of type	
++	increment	right to left
- -	dcrement	
~	complement	
!	not	
-	unary minus	
+	unary plus	
&	address of	
*	dereference	
*	multiply	left to right
/	divide	
%	modulus (remainder)	
+	add (binary)	left to right
-	subtract (binary)	
<<	shift left	left to right

>>	shift right	
<	less than	
<=	less than or equal to	
>	greater than	
>=	greater than or equal to	
==	equal	left to right
!=	not equal	
&	bitwise AND	
^	bitwise XOR (exclusive OR)	
|	bitwise OR	
&&	logical AND	
||	logical OR	
?:	conditional expression	
=	simple assignment	right to left
*=	multiply and assign	
/=	divide and assign	
%=	modulus and assign	
+=	add and assign	
=	subtract and assign	
<<=	left shift and assign	
>>=	right shift and assign	
&=	bitwise AND and assign	
|=	bitwise OR and assign	
^=	bitwise XOR and assign	
,	comma	left to right

Type conversions

The general rule for type conversion is to convert a "narrower" operand into a "wider" one without losing information, such as converting an integer into a long integer. The conversion from **char** to **long** is sign extension. Explicit type conversion can be forced in any expression, with a unary operator called a cast. In the example:

```
    (type-name) expression
```

the *expression* is converted to the named type

Program Control Flow

The statements in this section are used to control the flow of execution in a program. The use of relational and logical operators with these control statements and how to execute loops are also described.

if-else statement

- Syntax

  if *(expression)*
  statement1;
  [**else**
  statement2;

]

- Description

The **if-else** statement is a conditional statement. The block of statements executed depends on the result of the condition. If the result of the condition is nonzero, the block of its associated statements is executed. Otherwise, the block of statements associated with the **else** statement is executed if the **else** block exists. Note that the **else** statement and its block of statements may not exist as it is optional.

- Example

```
if (word_count > 80)
{
   word_count=1;
   line++;
}
else
   word_count++;
```

for statement

- Syntax

  for *(initial-expression; condition-expression;*
  update-expression) statement;

The *initial-expression* is executed first and only once. It is used to assign an initial value to a loop counter variable. This loop counter variable must be declared before the **for** loop. The *condition-expression* is evaluated prior to each execution of the loop. If the *condition-expression* is evaluated to be nonzero, the statement in the loop is executed. Otherwise, the loop exits and the first statement encountered after the loop is executed next. The *update-expression* executes after the statement of the loop.

- Description

The **for** statement is used to execute a statement or block of statements repeatedly.

- Example

```
for (i=0;i<10;i++)
   a[i]=b[i]; // copy elements from an array
                 // to another array
```

while statement

- Syntax

  while *(condition-expression)*
  statement;

- Description

The **while** statement is another kind of loop. When the *condition- expression* is nonzero, the while loop executes the *statement*. The *condition-expression* is checked prior to each execution of the *statement*.

- Example

```
i=0;
while (b[i] !=0)
{
   a[i]=b[i];
   i++;
}
```

do-while statement

- Syntax

  do

  *statemen*t;

  while *(condition-expressio*n);

- Description

  The **do-while** statement is another kind of while loop. The *statement* is always executed before the *condition-expression* is evaluated. Hence, the *statement* executes at least once, then checks the *condition-expression*.

- Example

```
i=0;
do
{
   a[i]=b[i];
   i++;
}while (i<10);
```

break and continue statement

- Syntax

  break;

  continue;

- Description

  The **break** statement is used to force an immediate exit from **while, for**, **do-while** loops and **switch**. The **break** statement bypasses normal termination and returns control to the previous nesting level if a **break** occurs within a nested loop.

  The **continue** statement orders the program to skip to the end of the loop and begins the next iteration of the loop. In the **while** and **do-while** loops, the **continue** statement forces the *condition-expression* to be executed immediately. In the **for** loop, control passes to the *update-expression*.

- Example

```
char a[10],b[10],i,j;
for (i=j=0;i<10;i++)// copy data from b[ ] to a[ ],
                    // skip blanks
{
   if (b[i]==0) break;
   if (b[i]==0x20) continue;
   a[j++]=b[i];
}
```

goto statement and label

- Syntax

  goto *label;*

- Description

  A *label* has the same form as a variable name, but followed by a colon. The scope of a label is the entire function.

- Example

  See the **switch** statement example

switch statement

- Syntax

```
switch (variable)
{
    case constant1:
        statement1;
        break;
    case constant2:
        statement2;
        goto Label1;
    case constant3:
        statement3;
        break;
    default:
        statement;
Labell: statement4;
        break;
}
```

The **switch** variable is tested against a list of constants. When a match is found, the statements with that constant are executed until a **break** statement is encountered. If no **break** statement exists, execution flows through the rest of the statements until the end of the **switch** routine. If no match is found, the statements associated with the **default** case are executed. The **default** case is optional.

- Description

The **if-else** statement can be used to select between a pair of alternatives, but becomes cumbersome when many alternatives exist. The **switch** statement is an alternative multi-way decision method that evaluates if an expression matches one of many alternatives, and branches accordingly. It is equivalent to multiple **if-else** statements. The **switch** statement's limitation is that the **switch** variable must be an integral data type, and can only be compared against constant values.

- Example

```
for (i=j=0;i<10;i++)
{
    switch (b[i])
    {
        case 0: goto outloop;
        case 0x20:break;
        default:
              a[j]=b[i];
              j++;
              break;
    }
}
outloop:
```

Functions

In the C language, all executable statements must reside within a function. Before a function is used or called, it must be either defined or declared, otherwise a warning message will be issued by the C compiler. Two syntax forms, namely classic and modern, are supported for function declaration and definition. Unlike the variable, there is no need and no way to assign a function in a specific bank for the MCU having multi-bank of ROM. The linker will locate functions into a appropriate ROM bank.

Classic form

return-type function-name (arg1, arg2,...)
var-type arg1;
var-type arg2;

Modern form

return-type function-name (var-type arg1, var-type arg2, ...)

In both forms, the *return-type* is the data type of the function returned value. If functions do not return values, then *return-type* must be declared as **void**. The *function-name* is the name of this function and is equivalent to a global variable of all other functions. The arguments, *arg1, arg2* etc, are the variables to be used in this function. Their data type must be specified. These variables are defined as formal parameters to receive values when the function is called.

➔ Function declaration

```
// classic form
return-type function-name (arg1, arg2, ...);
// modern form
return-type function-name (var-type arg1, var-type arg2,...);
```

➔ Function definition

```
// classic form
return-type function-name (arg1, arg2, ...)
var-type arg1;
var-type arg2;
{
statements;
}
// modern form
return-type function-name (var-type arg1, var-type arg2, ...)
{
statements;
}
```

➔ Passing arguments to functions

There are two methods for passing arguments to functions.

- Pass by value.

  This method copies the argument values to the corresponding formal parameters of the function. Any changes to the formal parameters will not affect the original values of the corresponding variables in the calling routine.

- Pass by reference.

  In this method, the address of the argument is copied to the formal parameters of the function. Within the function, the formal parameters can access the actual variables within the calling routine. Hence, changes to the formal parameters can be made to the variables.

➔ Returning values from functions

By using the **return** statement, a function can return a value to the calling routine. The returned value must be of a data type specified within the function definition. If *return-type* is **void**, it means no return value, therefore no value should be in the **return** statement. When a **return** statement is encountered, the function returns immediately to the calling routine. Any statements after the **return** statement are not executed.

Pointers and Arrays

Pointers

A pointer is a variable that contains the address of another variable. For ex-ample, if a pointer variable, namely varpoint, contains the address of a variable var, then varpoint points to var. The syntax to declare a pointer variable is

*data-type *var_name;*

The *data-type* of a pointer is a valid C data type. It specifies the type of variable that *var_name* points to. The asterisk (*) prior to *var_name* tells the C compiler that *var_name* is a pointer variable. Two special operators, the asterisk (*) and ampersand (&), are associated with pointers. The address of a variable can be accessed by preceding this variable with the & operator. The * operator returns the value stored at the address pointed to by the variable.

In addition to * and &, there are four operators that can be applied to the pointer variables: +, ++, -, --. Only integer quantities may be added or subtracted from pointer variables. An important point to remember when performing pointer arithmetic is that the value of the pointer is adjusted according to the size of the data type it is pointing to.

Arrays

An array is a list of variables that are of the same type and which can be referenced by the same name. An individual variable in the array is called an array element. The first element of an array is defined to be at an index of 0 and the last element is defined to be at an index of the total elements minus one. C stores one-dimensional arrays in contiguous memory locations. The first element is at the lowest address. C does not perform boundary checking for arrays.

Assignment from an entire array to another array is not allowed. To copy, each individual element must be copied one by one from the first array into the second array. Any array element can be used anywhere a variable or a constant can be used.

Structures and Unions

Structures

- Syntax

```
struct struct-name
{
    data-type member1;
    data-type member2;
    ...
    data-type membern;
} [ variable-list ] ;
```

- Description

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures may be copied and assigned to, passed to functions and returned by functions. C allows bit fields. Nested structures are also allowed.

The reserved word **struct** indicates a structure is to be defined while *struct-name* is the name of the structure. Within the structure, *data-type* is one of the valid data types. Members within the structure may have different data types. The *variable-list* declares variables of the type *struct-name*. Each item in the structure is referred to as a member.

After defining a structure, other variables of the same type are declared with the following syntax:

struct *struct-name variable-list;*

To access a member of a structure, specify the name of the variable and the name of member separated by a period. The syntax is

variable.member1

where *variable* is the variable of structure type and *member1* is a member of the structure. A structure member can have a data type with a previously defined structure. This is referred to as a nested structure.

- Example

```
struct person_id
{
   char id_num[6];
   char name[3];
   unsigned long birth_date;
} mark;
```

Unions

- Syntax

union *union-name*
{
data-type member1;
data-type member2;
...
data-type memberm;
} [*variable-list*] ;

- Description

Unions are a group of variables of differing types that share the same memory space. A union is similar to a structure, but its memory usage is very different. In a structure, all the members are arranged sequentially. In a union, all members begin at the same address, making the size of the **union** equal to the size of the largest member. Accessing the members of a union is the same as accessing the members of a structure. **union** is a reserved word and *union-name* is the name of the union. The *variable-list*, which is optional, contains the variables that have the same data type as *union-name.*

- Example

```
union common_area
{
   char name[3];
```

```
        int id;
        long date;
    } cdata;
```

Preprocessor Directives

The preprocessor directives give general instructions on how to compile the source code. It is a simple macro processor that conceptually processes the source codes of a C program before the compiler properly parses the source program. In general, the preprocessor directives do not translate directly into executable code. It removes preprocessor command lines from the source file and expands macro calls that occur within the source text and adds additional information, such as the **#line** command, on the source file. The preprocessor directives begin with the # symbol. A line that begins with a # is treated as a preprocessor command, and is followed by the name of a command. The following are the preprocessor directives:

Macro substitution: #define

- Syntax

  #define *name replaced-text*
  #define *name [(parameter-list)] replaced-text*

- Description

The **#define** directive defines string constants that are substituted into a source line before the source line is evaluated. The main purpose is to improve source code readability and maintainability. If the replaced-text requires more than one line, the backslash (\) is used to indicate multiple lines.

- Example

```
#define TOTAL_COUNT    40
#define USERNAME       "Henry"
#define MAX(a,b)       (((a)>(b))?(a):(b))
#define SWAP(a,b)      {int tmp; \
                         tmp=a; \
                         b=a; \
                         a=tmp;}
```

#error

- Syntax

  #error   message-string

- Description

The **#error** directive generates a user-defined diagnostic message, *message-string.*

- Example

```
#if  TOTAL_COUNT > 100
#error  "Too many count."
#endif
```

Conditional inclusion: #if #else #endif

- Syntax

  #if *expression*
  source codes1
  [#**else**
  source codes2]
  #endif

- Description

The **#if** and **#endif** directives pairs are used for conditionally compiling code depending upon the evaluation of the *expression*. The #**else** which is optional provides an alternative compilation method. If the *expression* is nonzero, then the *source codes1* will be compiled. Otherwise, the *source codes2*, if it exists, will be compiled.

- Example

```
#define MODE 2
#if MODE > 0
  #define DISP_MODE MODE
#else
  #define DISP_MODE 7
#endif
```

Conditional inclusion : #ifdef

- Syntax

  #ifdef *symbol*
  source codes1
  [**#else**
  source codes2]
  #endif

- Description

The **#ifdef** directive is similar to the **#if** directive, except that instead of evaluating the expression, it checks if the specified *symbol* has been defined or not. The **#else** which is optional provides alternative compilation. If the *symbol* is defined, then the *source codes1* will be compiled. Otherwise, the *source codes2*, if it exists, will be compiled.

- Example

```
#ifdef DEBUG_MODE
#define TOTLA_COUNT 100
#endif
```

Conditional inclusion : #ifndef

- Syntax

  #ifndef *symbol*
  source codes1
  [**#else**
  source codes2]
  #endif

- Description

The **#ifndef** directive is similar to the **#ifdef** directive. The **#else** which is optional provides alternative compilation. If the *symbol* has not been defined, then the *source codes1* will be compiled. Otherwise, the *source codes2*, if it exists, will be compiled.

- Example

```
#ifndef DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

Conditional inclusion : #elif

- Syntax

  #if *expression1*
  source codes1
  #elif *expression2*
  source codes2
  [**#else**
  source codes3]
  #endif

- Description

  The **#elif** directive is accompanied with the **#if** directive. It provides other compilation conditions in addition to the usual two. If the *expression1* is nonzero, then the *source codes1* will be compiled. If *expression1* is zero, then *expression2* is checked to see if it is nonzero. If so then the *source codes2* will be compiled. Otherwise, the *source codes3*, if it exists, will be compiled.

- Example

```
#if  MODE==1
#define  DISP_MODE 1
#elif  MODE==2
#define  DISP_MODE 7
#endif
```

Conditional inclusion : defined

- Syntax

  #if defined *symbol*
  source codes1
  [**#else**
  source codes2]
  #endif

- Description

  The unary operator **defined** can be used within the directive **#if** or **#elif**.

  A control line of the form

  #ifdef symbol

  is equivalent to

  #if defined symbol

  A line of the form

  #ifndef symbol

  is equivalent to

  #if !defined symbol

- Example

```
#if defined DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

#undef

- Syntax

  #**undef** *symbol*

- Description

  The **#undef** directive causes the *symbol's* preprocessor definition to be erased. Once defined, a preprocessor symbol remains defined and in scope until the end of the compilation unit or until it is undefined using an **#undef** directive.

- Example

```
#define  TOTAL_COUNT 100
...
#undef  TOTAL_COUNT
#define  TOTAL_COUNT 50
```

File inclusion: #include

- Syntax

  #include <*file-name*>
  or
  #include ”*file-name”*

- Description

  #include inserts the entire text from another file at this point in the source file. When <*file-name*> is used, the compiler looks for the file in the directory specified by the environment variable INCLUDE. If the INCLUDE is not defined, the C compiler looks for the file in the path. When ”*file-name”* is used, the C compiler looks for the file as specified. If no directory is specified, the current directory is checked.

- Example

```
#include <ht48c10-1.h>
#include "my.h”
```

Language Extensions and Restrictions

Holtek C language provides a number of extensions for ANSI C. Most of these provide support for elements of the Holtek microcontroller architecture. Due to the limited resource of the microcontroller, there are also some restrictions you should take care.

Keywords

The following is a list of the keywords available in Holtek C.

@	bit	norambank	rambank0	vector

The following keywords and qualifiers are not supported:

double	float	register

Memory bank

For variables located in high banks (not bank 0), they should be accessed through indirect addressing mode. To achieve the efficiencies, you might locate the most used variables in Ram bank 0. The Holtek C provides you a **rambank0** keyword to declare variables in bank 0.

- Syntax

  #pragma rambank0
  //data declarations
  #pragma norambank

- Description

  The **rambank0** keyword directs the compiler to declare subsequent variables to locate in Ram bank 0 until the **norambank** keyword meets. For the single Ram bank

MCU, these two keywords will be ignored.

- Example

```
#pragma rambank0
unsigned int i, j;        //i, j located in Ram bank 0
long len;                 //len located in Ram bank 0

#pragma norambank
unsigned int iflag;       //iflag's bank number is unknown

#pragma rambank0
int tmp;                  //tmp located in Ram bank 0

…
i = 1;              //MOV A,1
                    //MOV _i, A

iflag = 1;          //MOV A,BANK _iflag
                    //MOV [04H],A
                    //MOV A,OFFSET _iflag
                    //MOV [03H],A
                    //MOV A,1
                    //MOV [02H],A
```

Bit data type

Holtek C provides you with a bit data type which may be used for variable declarations, argument lists, and function return values. A bit variable is declared just as other C data types are declared. For the MCU supports multiple RAM banks, you should declare the bit variables in the RAM bank 0 (using **#pragma rambank0**) area.

- Example

```
#pragma rambank0
bit test_flag;      //bit var should locate in rambank0

bit testfunc(       //bit function
 bit f1,            //bit arguments
 bit f2)
{
       ...
       return 0;    //return bit value
}
```

- Restriction
  - — To get the benefit of the bit data type, it is not recommended to declare a bit array variable.
  - — There is no bit pointer.
  - — There is no bit data type member in a structure declaration.

Inline assembly

- Syntax

  #asm
  < [*label*:] *opcode* [*operands*] >
  ...
  #endasm

- Description

The **#asm** and **#endasm** are the inline assembly preprocessor directives. The **#asm** directive inserts Holtek's assembly instruction(s) after this directive (or within the directive **#asm** and directive **#endasm**) into the output file directly.

- Example

```
// convert low nibble value in the accumulator to ASCII
#asm
; this is an inline assembly comment
and a, 0fh
sub a, 09h
sz c
add a, 40h-30h-9
add a, 30h+9
#endasm
```

Interrupts

The Holtek C language provides a means for implementing interrupt service routines (ISRs) through the preprocessor directive #**pragma**. The directive **#pragma vector** is used to declare the name and address of the ISRs. Any function declared later with the same name as defined with **#pragma vector** is the ISR for the vector. The return statement within the ISR generates a **RETI** instruction.

- Syntax

  #**pragma vector** *symbol* **@** *address*
- Description

symbol is the name of the interrupt service routine.
address is the interrupt address. The reset vector (address 0) is reserved for *main* function and therefore cannot be used.

- Restriction

There are five restrictions to keep in mind when writing an ISR.

— There is no parameter for ISR; the return type is **void**.

— The ISR is not reentrant. Do not enable the interrupt in the ISR.

— Do not call the ISR explicitly in your programs. It should always be invoked implicitly by the system while the interrupt coming.

— Do not call the user defined function written in C within the ISR. It is safe to use the system calls. If a function has to be called within the ISR then it should be written in assembly. It is safety to call the built-in functions in the ISR.

— It is the user's responsibility to preserve the affected registers when they are used in inline assembly in the ISR. The Holtek C compiler will only preserve the affected registers written in the C statements.

- Example

```
#pragma vector timer0 @ 0x8
extern void ASM_FUNCTION();
void setbusy(){
...
}

void timer0(){
  ...
  ASM_FUNCTION();     //The ASM_FUNCTION should be
                      // an assembly function
  _delay(3);         //OK. built-in function

  setbusy();         //Wrong! do not call C function
}
```

Variables

The operator '**@**' can be used to specify the address of variables in the data memory.

- Syntax
  data_type   varaible_name **@** *memory_location*
- Description

The *memory_location* specifies the address variable located. For single bank of RAM/ROM, the *memory_location* is one byte. For multiple banks of RAM/ROM, the *memory_location* is two bytes, the high byte is the bank number. You should check the data sheet of the Holtek MCUs to get the information of the available RAM space.

- Example

```
int v1 @ 0x5B; // declare v1 in the RAM bank 0 offset 0x5B
int v2 @ 0x2F0;//declare v2 in the RAM bank 2 offset 0xF0
```

Static Variables

Holtek C supports file scope static variables while local static variables does not.

- Example

```
static int i; // file scope static
void f1(){
       i = 1;  //OK
}
void f2(){
       static int j; //Wrong. local static variable
                     // is not supported
       ...
}
```

Constants

Holtek C supports binary constants. Any string that begins with 0b or 0B will be treated as a binary constant.

For example:

```
0b101= 5
0b1110= 14
```

Functions

Avoid using reentrant and recursive code.

Function can not return a structure data type.

Arrays

An array should be located in a contiguous block of memory and must not have more than 256 elements. To speak precisely, the size of an array is limited to the size of the

RAM bank of the Holtek MCU you used.

Constant variables

Constant variables must be declared in global scope and be initialized when declared. A constant variable could not be declared as external.

A constant array should specify the array size otherwise an error generated. The size of an array is limited to 255 bytes.

```
const char carray[]= {1,2,3}; //wrong
const char carray[3]= {1,2,3}; //right
```

A constant string must be used in the C file with the main function.

```
//test.c
char *str;
void f1(char *s);
void f2(){
   f1(“abcd”); //”abcd” is a constant string
              // If there is no main() function declared
              // in test.c then the Holtek C compiler would
              // generate an error.
   str = “1234”; //”1234” is constant string
}
...
void main(){
   ...
}
```

Pointer

Pointer cannot be applied to constant and bit variables

Initial value

Global variables cannot be initialized when declared. Local variables do not have this constraint. Constant variables must be initialized when declared.

For example:

```
unsigned int i1= 0;//illegal declaration; can not be
                   //initialized
unsigned int i2;
const unsigned int i3; //illegal declaration; should be
                   //initialized
const unsigned int i4=5;
const char a1[5];  //illegal declaration; should be
                   //initialized
const char a2[5]={0x1,0x2,0x3,0x4,0x5};
const char a3[4]="abc"; //={'a', 'b', 'c', 0}
const char a4[3]="abc"; //={'a', 'b', 'c'}
//const char a5[2]=”abc”; //array size mismatched
```

Multiply/Divide/Modulus

The multiply, divide and modulus ("*", "/", "%") operators are implemented by system calls.

Built-in Function

- WDT & halt & nop

C system call	Assembly code
void _clrwdt( )	CLR WDT
void _clrwdt1( )	CLR WDT1
void _clrwdt2( )	CLR WDT2
void _halt( )	HALT
void _nop( )	NOP

- Rotate right/left

```
void _rr(int*);   //rotate 8 bits data right
void _rrc(int*);  //rotate 8 bits data right through carry
void _lrr(long*); //rotate 16 bits data right
void _lrrc(long*);//rotate 16 bits data right through carry
void _rl(int*);   //rotate 8 bits data left
void _rlc(int*);  //rotate 8 bits data left through carry
void _lrl(long*); //rotate 16 bits data left
void _lrlc(long*);//rotate 16 bits data left through carry
```

- swap nibble

```
void _swap(int *); //swap nibbles of 8 bits data
```

- delay cycle

```
void _delay(unsigned long); //delay n instruction cycles
```

The _delay function forces the MCU to execute the specified cycle count. A value of zero causes an endless loop. The parameter of the _delay could be constant value only. It does not accept a variable.

Example :

```
//assume the watch dog timer is enable
//and using one instruction

void error(){
      _delay(0); //infinite loop, same as while(1);
}

void dotest(){
      unsigned int ui;
      ui = 0x1;
      _rr(&ui);   //rotate right
      if (ui != (unsigned int)0x80) error();
      ui = 0xab;
      _swap(&ui);
      if (ui != (unsigned int)0xba) error();
}

void main(){
       unsigned int i;
       for(i=0; i<100; i++){
```

```
                    _clrwdt();
                    _delay(10); //delay 10 instruction cycle
                    dotest();
              }
       }
```

Example :

```
       //assume the watch dog timer is enable
       //and using two instructions
       void dotest(){
              ...
       }

       void main(){
              unsigned int i;
              for(i=0; i<100; i++){
                    _clrwdt1();
                    _clrwdt2();
                    dotest();
              }
       }
```

Stack

Because the Holtek micro-controllers have limited depth stack the programmer needs to consider the function call depth to avoid stack overflow. The multiply, divide, modulus, and const variables are implemented by "call" instructions, taking one stack.

Operator/System Function	**Stack Needed**
main ( )	0
_clrwdt( )	0
_clrwdt1( )	0
_clrwdt2( )	0
_halt( )	0
_nop( )	0
_rr(int*)	0
_rrc(int*)	0
_lrr(long*)	0
_lrrc(long*)	0
_rl(int*)	0
_rlc(int*)	0
_lrl(long*)	0
_lrlc(long*)	0
_swap(int*)	0
_delay(unsigned long)	1
*	1
/	1
%	1
constant array	1

Chapter 2
Mixed language

2

The Holtek Cross Tools (Cross Assembler, Cross Linker, Library and Holtek C compiler) provide methods to program with mixed languages, Holtek assembly language and C language. That means a project can consist of source files programming with assembly language and C language. However, the programmer should conform to some rules when programming with mixed language. In order to facilitate the program coding, this chapter describes the conventions that Holtek C compiler compiles a C program into the assembly language, how to define the subroutine name, etc. The following are the topics:

- Little endian
- Naming rule of functions and parameters
- Parameter passing
- Return value
- Preserving registers
- Calling assembly function from C program
- Calling C function from assembly program
- Programming ISR with assembly language

Little endian

The data format adopted by the Holtek C compiler is Little-Endian, i.e. the low byte of a WORD is the WORD's least significant byte, and the high byte is the most significant. In memory allocation, the low byte occupies the lower address and high byte occupies the higher address.

For example

```
long var @ 0x40;
var = 0x1234;
```

Then the address 0x40 contains 0x34, and the address 0x41 contains 0x12.

Naming rule of function and parameters

The Holtek Cross Assembler is **non case-sensitive** when handling symbol names. Actually, all symbol names are translated into uppercase no matter what the original form is. But the Holtek C language is **case-sensitive**. Due to the difference of these two languages, the variables and functions which are defined in C source files and referred by the assembly program should be defined as uppercase.

The names of the global variables and functions in C language are prefixed with underscore when C compiler translates them into the assembly language. For the local variables, if a local variable is declared without referenced, the C compiler won't reserve memory space for it. By checking the assembly file generated by the Holtek C compiler, the programmer can find out what the translated name of the C local variable is.

Global variable

A global variable in a C file is translated into the same case letters with a prefixed underscore.

For example,

```
TimerCt
TMP
```

will be translated into

```
_TimerCt
_TMP
```

Local variable

If a local variable in a C function is not referenced by other programs, it will not be translated into assembly language. By checking the assembly file to find out what the result is.

```
void main(){
  int i, j, k;         // k is not used
  long m;
  char c;
  i = j = m = c = 2;
  #asm
    set CR3[1].2  ;set bit 10 of m => m |= 0x400
  #endasm
}
```

The corresponding part of the assembly file will looks like this:

```
#line 2 "C:\HT-IDE2000\SAMPLE\NAME.C"
LOCAL CR1 DB ? ; i
#pragma debug variable 2 CR1 i
#line 2 "C:\HT-IDE2000\SAMPLE\NAME.C"
LOCAL CR2 DB ? ; j
#pragma debug variable 2 CR2 j
#line 3 "C:\HT-IDE2000\SAMPLE\NAME.C"
LOCAL CR3 DB 2 DUP (?) ; m
#pragma debug variable 2 CR3 m
#line 4 "C:\HT-IDE2000\SAMPLE\NAME.C"
LOCAL CR4 DB ? ; c
#pragma debug variable 2 CR4 c
```

The second and third line indicates that the *i* is translated into *CR1* in the assembly file. By the same way, *j* is translated into *CR2*, *m* is *CR3* and *c* is *CR4*. The *k* is not referenced so it is not translated.

Caution: If the local variables are added to or removed from or arranged the order, then the translated names might be changed by the C compiler.

For the above sample code, if the micro controller supports multiple RAM banks, then the instruction

```
set CR3[1].2
```

can execute correctly or not. The program will corrupt if the CR3 is allocated in high bank. But this phenomenon won't happen, because the local variable is defined with **LOCAL** directive in the translated assembly file and instructs the assembler to allocate the variable in the RAM bank 0. Hence it can execute correctly in the way like a variable does in the single RAM bank.

Function

Like the global variable, a function in a C file is translated into the same case letters with a prefixed underscore.

For example,

```
GetKey
IsBusy
```

will be translated into

```
_GetKey
_IsBusy
```

Function parameters

The names of the function parameters in a C file are translated into the function name following the number of the parameters occurring, indexed from 0.

For example,

```
GetKey(int row, long col)
```

row is translated into GetKey0
col is translated into GetKey1

Parameter passing

Due to the micro controller resource's limitation, the Holtek C compiler passes parameters to function via the RAM space instead of the stack. The naming of the function parameters are the function name appending the number of the parameters occurring, indexed from 0. Like the local variable, the function parameters are also allocated in the RAM bank 0.

For example:

```
void    function (int a, int b)
```

Then the parameter *a* will be translated into *function0, b* will be *function1.*

For mixed language, the data type of function parameters should always declares as BYTE in assembly, if it's more than one byte, e.g. WORD (2 bytes), programmer should use the instruction "DB n DUP(?) " to declare it.

Return value

The return value of a C function is located in the **A** register or in the **RH** system variable. If the size of the return value is one byte (e.g. **char**, **unsigned char**, **int**, **unsigned int**, **short**, **unsigned short**), then the value is stored in the **A** register. If it is two bytes (e.g. **long**, **unsigned long**, pointer), then the high byte is stored in the **RH** and the low byte is stored in the **A** register.

Note: The **RH** variable is located in RAM bank 0.

Preserving registers

Except the ISR, there is no need to preserve the registers when writing a function in assembly. If a user writes an ISR in assembly language, then it is his responsibility to preserve the registers used in the ISR.

Calling assembly function from C program

This section describes the steps to call an assembly function from a C program. The steps are divided into two parts, one is for the assembly file, the other is for C file.

➔ **In Assembly File**

- Declare **RH** as external byte variable if the return value is two bytes.
- Declare the function name with prefixed underscore as public.
- Declare the function parameters, if there is, in the RAM bank 0 as public. Be aware of the naming of parameters.
- Put the return values into **A** or **RH**.

➔ **In C File**

- Declare the prototype of the external function name with uppercase
- Call it

Example

The following function is defined in assembly file and called by a C program,

```
long    KEYIN(int row, long col);
```

In assembly file

```
;;Declare external byte variable RH
EXTERN RH:BYTE

;;Declare function name & parameters as public
PUBLIC _KEYIN, KEYIN0, KEYIN1

;;Declare parameters
RAMBANK 0 KEYINDATA ;suppose the MCU has multiple ram banks
KEYINDATA .section 'data'
KEYIN0 DB ?                 ;row
KEYIN1 DB 2 DUP (?)          ;col, don't use “KEYIN1 DW ?”

;function body
CODE .section 'code'
_KEYIN:
```

```
. . .
   MOV A, KEYIN0              ;retrieve row
. . .
   MOV A, KEYIN1              ;retrieve low byte of col
. . .
   MOV A, KEYIN1[1]          ;retrieve high byte of col
. . .
;; Put the return values into A and RH
   MOV A,0A0H                   ;suppose the return value is 0xA010
   MOV RH,A                     ; store high byte 0xA0 to RH
   MOV A,10H                    ; store low byte 0x10 to A
   RET
```

In C file

```
// Declare the external function name with uppercase
extern long KEYIN(int row, long col);
long rc;
. . .
// Call it
   rc = KEYIN(10, 20L);
```

Calling C function from assembly program

This section describes the steps to call a C function from an assembly program. For the micro controller with multiple ROM banks, it is important to set the BP (bank pointer) before calling the function.

➔ **In C File**

- Declare the function name with uppercase

➔ **In Assembly File**

- Declare **RH** as external byte variable if the return value is two bytes.
- Declare the external function name with prefixed underscore
- Declare the function parameters as external if there is. Be aware of the naming of parameters.
- Set function parameters if there is
- Call C function

  Call the C function directly if the micro controller supports single RAM/ROM bank.

  Set BP to the bank of function first, then calls the C function if the micro controller supports multiple RAM/ROM banks.
- Get return value from **A** or **RH**

Example 1

The following function is defined in C language and called by assembly program

```
long   KEYIN(int row, long col);
```

and the micro controller has single ROM bank.

```
//----------------------------------------
// In C file, function definition
//----------------------------------------
long    KEYIN(int row, long col){
. . .
}


;;---------------------------
;; In assembly file
;; ---------------------------
;;Declare external byte variable RH
EXTERN RH:BYTE

;;Declare the external function name with prefixed underscore
extern _KEYIN: near              ;; underscore and function name

;;Declare the function parameters as external variables
extern KEYIN0:byte               ; function parameter : row
extern KEYIN1:byte               ; function parameter : col, although it's 2 bytes,
                                  ; to declare it as BYTE
code_ki .section 'code'

;; Set function parameters for calling    KEYIN(0x10, 0x200L)
mov a,10H
mov KEYIN0,a                 ; put value to function parameter : row
mov a,2H
mov KEYIN1[1],a               ; put value to high byte of parameter : col
clr KEYIN1                   ; put value 0 to low byte of parameter : col

;; Call C function
call _KEYIN

;; Get return value from A or RH
;; A register keeps low byte of return value
;; RH keeps high byte of return value
```

Example 2

The following function is defined with C language and called by assembly program,

long   KEYIN(int row, long col);

and the micro controller supports multiple ROM banks

```
//--------------------------------------
// In C file
//--------------------------------------
long KEYIN(int row, long col){
. . .
}


;--------------------------
; In assembly file
```

```
;----------------------------

;;Declare external byte variable RH
EXTERN RH:BYTE

;; Declare the external function name with prefixed underscore
extern _KEYIN:near

;; Declare the function parameters as external variables
extern KEYIN0:byte                ; parameter   : row
extern KEYIN1:byte                ; parameter : col, although it's 2 bytes,
                                  ; only declare one BYTE

code_ki   .section   'code'

;;Set function parameters for calling   KEYIN(0x10, 0x200L)
mov   a,10
mov   KEYIN0,a                   ; parameter : row
mov   a ,2
mov   KEYIN1[1],a                ; high byte of the parameter : col
clr    KEYIN1                    ; low byte of the parameter : col

;; Call C function in multiple ROM banks
;; Set BP to the bank of function first

mov   a, bank _KEYIN
mov   bp , a                     ; change the bank number
call    _KEYIN

;; Get return value from A or RH
;; A register keeps low byte of return value
;; RH keeps high byte of return value
```

Programming ISR with assembly language

An ISR (Interrupt Service Routine) is invoked by hardware interrupt. It should not be explicitly called by user, hence it doesn't have parameters passing nor return value.. When you write an ISR in assembly, there is nothing to do with the other c files. All you need to do is to add the assembly file into the project. Please refer to the assembly language user's guide for more information about ISR programming.

Do not call a C function from an ISR, no matter the ISR is written in assembly or C.

Chapter 3 Programming With C Language

3

This chapter covers the following sections:

Start a C Program

The source files of a project in HT-IDE3000 may be written by Holtek assembly language or C language. After a chip reset, the program always begins execution at address 0 of Program Memory. When there is at least one source file is programming with C language and the program entry point is the C program, then the function *main* is forced to be located on address 0 of Program Memory by C compiler. Also, the following rules must be followed,

- A *main* function has been defined in a source file and all other source files can not define it. The *main* function is the entry point of program execution.
- The address 0 of Program Memory can not be used for other functions, tables or Code sections . This address is used by *main* function only. The following statement will cause an error when building the project.

  #**pragma** **vector** ResetFunction @ 0x00

  The function ResetFucntion can not be defined at address 0

Example

```
void test(){
}

void main(){          // define the main function
     test();
}
```

Define the Interrupt Service Routines

When a project needs to handle the MCU interrupts and the corresponding interrupt service routines (ISR) are going to program with C language, then the proper usage and restrictions should be noticed.

It is not necessary to preserve the system registers explicitly, the Holtek C compiler preserves the used registers automatically.

Declare the name and address of ISR in C source file

To declare the name and address by using the **pragma vector** statement as follow,

#**pragma** **vector** *IsrRoutineName* **@** *address*

pragma and **vector** are keywords.
IsrRoutineName is the name of the interrupt service routine.
address is the memory address of the interrupt service routine. The address 0 is reserved for the *main* function and cannot be used.

Define the ISR in C source file

To define a function with the same name as *IsrRoutineName* in above **#pragma** vector.

```
#pragma   vector   _ExternISR   @   0x04

void _ExternISR(void){

}
```

Restriction

There are some restrictions to keep in mind when writing an ISR with C language.

- There is no parameter for ISR and the return type is **void**.
- The ISR is not reentrant. Do not enable the interrupts in ISR.
- Do not call the ISR explicitly within the programs. It should always be invoked implicitly by the system when the interrupt occurs.
- Do not call any user defined C function in ISR. But calling the built-in functions is safe. If ISR wants to call a function, the function should be written with assembly language.
- If ISR contains inline assembly instructions, then the affected registers due to these instructions execution should be preserved before execution and restored after execution . The Holtek C compiler only preserves the affected registers caused by C statements.

Example

```
#include <ht47c20.h>
#pragma      vector _ExternISR    @ 0x4
#pragma      vector _TimeBaseISR   @ 0x8
```

```
#pragma      vector _RTCISR        @ 0xc
#pragma      vector _TimerISR     @ 0x10

unsigned count;

void _ExternISR(void){
}

void _TimeBaseISR(void){
   count=count>>7|count<<1;
   _pa = count;
}

void _RTCISR(void){
}

void _TimerISR(void){
}

void main()
{
   count=0xee;
   _intc0=0x05;    //EMI&ETBI ENABLE
   while(1);
}
```

Define Tables and Symbols in Program Memory

The Holtek C compiler allocates program memory for those tables and symbols with fixed value. When a table (array) or a symbol with the fixed value, then it can be located in program memory by declaring its type as **const.** The usage of this symbol or array is the same as those symbols or array defined in Data Memory. The difference is the symbol or array in Program Memory can not be modified. The maximum value of these constants is 255, however if higher values are needed they can be separated into several constants.

Example

```
// below three variables are in Program Memory
const unsigned char ascii[16]="0123456789ABCDEF";
const unsigned char pattern[16]={0,1,2,3,4,5,6,7,8,
                               9,10,11,12,13,14,15};
const unsigned int cl = 0x8B;

// below variables are in Data Memory
#pragma rambank0
unsigned char str[2];

void itoa(unsigned int v, unsigned char *s){
  *s = ascii[v & 0xf];
  _swap(&v); //swap nibble
  *(s+1) = ascii[v & 0xf];
}

void main(){
  unsigned int val;
  val = cl;
  itoa(val, str);
}
```

Note: The symbols or array of type **const** should be initialized when declared. The size of a **const** array should also be specified.

Define Variables in Data Memory

There are two or three functional groups in Holtek MCU Data Memory, the register area, the general data area and the LCD data area. The Data Memory may consists of more than one bank, the RAM bank 0 and other RAM bank area.

The register area is the memory resident MCU function registers. The LCD data area stores the data which are used for LCD display. The general data area applies for the variables when programs execute. All Holtek's MCUs have the register area and the RAM bank 0 data area. But only some of them have more than one RAM bank. Please refer to corresponding data sheet for more information.

If not explicitly specified address, a variable will be allocated to the general data area by C compiler, it is relocatable. As for registers and the LCD data area, a proprietary address should be specified explicitly, otherwise it will be taken as a relocatable variable.

Specify Address to a Variable

The operator '**@**' is used to specify the address of variables in Data Memory.

- Syntax

  data_type   varaible_name   @   memory_address

- Description

  The *memory_address* specifies the address at which the variable *variable_name* is located. It is comprised of RAM bank number and the address within the RAM bank. The high byte of *memory_address* is the RAM bank number and the low byte is the address within this RAM bank. The *data_type* is the allowed the data type.

- Example

```
int v1 @ 0x50;  // v1 is in address 0x50 of RAM bank 0
int v2 @ 0x380; // v2 is in address 0x80 of RAM bank 3
int v3 @ 0xef0; // v3 is in address 0xf0 of RAM bank 14
```

Access Variables in Multiple RAM Banks

In assembly language it is necessary to set the bank pointer and use the indirect addressing mode to access the high RAM bank memory. Holtek C compiler has done these works automatically, users don't have to do any effort.

- Example

```
int v1 @ 0x5B;   // v1 is in address 0x5B of RAM bank 0
int v2 @ 0x2F0;  // v2 is in address 0xF0 of RAM bank 2
int v3;          // bank number is unknown

void main(){
   v1 = 10; //access bank 0 variable
   v2 = 10; //access high bank variable
   v3 = 10;
}
```

Specify Variables In RAM Bank 0 (Improve the Performance)

For MCU with multiple RAM banks, it must use indirect access instructions to access the variables in high RAM banks. The more of these accesses, the more instructions are executed. The result is to reduce the program performance. Hence, it had better to define those frequently used variables in RAM bank 0. By using C preprocessor **pragma rambank0,** the frequently used variables can be defined in RAM bank 0. It forces the Holtek Linker to find memory from the RAM bank 0 for the specified variables. If the RAM bank 0 has no enough memory to hold the variables, the Linker will issue an error even if there is enough memory in other bank. For this situation, user should rearrange the variables in the ***rambank0*** block. The preprocessor **pragma norambank** will end the **rambank0** function. All variables declared between ***rambank0*** and **norambank0** will be allocated to the RAM bank 0 unless RAM bank 0 has exhausted the space.

- Syntax

  #pragma rambank0
  //data declarations : variables defined in this block will be in RAM bank 0

  #pragma norambank
  //data declarations : variables defined in here are not necessary in RAM bank 0

- Description

  The **rambank0** keyword directs the compiler to declare subsequent variables to be located in the RAM bank 0 until the **norambank** keyword or end of file is encountered. For MCU with the single RAM bank, these two keywords will be ignored.

- Example

```
// default is norambank
unsigned int v1;         //v1's bank number is unknown

//switch to rambank0
#pragma rambank0
unsigned int i, j;       //i, j located at RAM bank 0
long len;                //len located at RAM bank 0

//In rambank0 area the address cannot be larger than 0x100
unsigned char uc0 @ 0x83;


// back to norambank
#pragma norambank
unsigned int iflag;     //bank number of iflag is unknown
unsigned char uc @ 0x140;

//switch to rambank0 linking mode
#pragma rambank0
bit bitflag;            //bit variable should always be
                        // declared in rambank0 block

void main(){
    i = 1;              //MOV A,1
                        //MOV _i, A
   iflag = 1;      //MOV A,BANK _iflag
                   //MOV [04H],A
                   //MOV A,OFFSET _iflag
                   //MOV [03H],A
                   //MOV A,1
                   //MOV [02H],A
   uc0 = uc = 0;
   bitflag = 1;
}
```

Pointer boundary

The address pointed by a pointer cannot cross the bank boundary while doing the pointer arithmetic. It will wrap when it overflows or underflows.

- Example :

```
#pragma rambank0
unsigned char *p1;
unsigned long *p2;

void main(){
   p1 = (unsigned char *)0x2f0;
   p1 += 0x20; //now p1 points to address 0x210, not 0x310

   p1 = (unsigned char *)0x100;
   p1--;  //now p1 points to address 0x1ff, not 0xff

   p2 = (unsigned long*)0x3fe;
   p2++; // 'long' occupies two bytes.
         // now p2 is pointed to 0x300 not 0x400.
}
```

Because a pointer cannot cross the bank boundary, the Holtek C does not support long integer pointer arithmetic.

.

- Example :

```
#pragma rambank0
unsigned char *p1, *p2;
unsigned int i;
unsigned long len;

void main(){
   p1 = p2+10; //ok
   p1 = p2+0x100; //error, 0x100 is a long integer
   p1 += i; //ok
   p1 += len; //error, len is a long integer
}
```

Access the LCD data area

Holtek C provides an easy way to access the LCD data area. To declare variables corresponding to LCD data memory address by using the '**@**' operator described in section **Specify Address To a Variable.** The following example demonstrates how to declare the LCD variables and how to access them.

.

- Example :

```
// LCD data memory is at RAM bank 14 (0x0e)
// lcd_day is at address 0x80 of RAM bank 14
// lcd_mon is at address 0x82 of RAM bank 14

#include <HTG2190.H>
// delcared lcd_day at LCD data area
unsigned char lcd_day @ 0xe80;

//declared lcd_mon at LCD data area
unsigned char lcd_mon @ 0xe82;

#pragma rambank0
unsigned int i, j;
unsigned char *lcd_ptr;

/*
Delcared non RAM bank 0 variables
A realistic scenario is that the variables are declared within the rambank0 block
if the memory is available.
*/

#pragma nonrambank0
unsigned int tmp;

void main(){
      lcd_mon = 0x10;              // put value 0x10 to LCD data memory 0xe82
      lcd_ptr = &lcd_day;          // lcd_ptr points to LCD data memory 0xe80
      * lcd_ptr = 0xff;            // put value 0xff to LCD data memory 0xe80
      *(lcd_ptr+1) = 0xa0;         // put value 0xa0 to LCD data memory 0xe81
}
```

MCU Special Function Registers

The Holtek MCU special function registers reside in the leading area of RAM bank 0. This data memory will not be used for general variables.

Access the function registers

To access the special function registers, it is necessary to bind a variable to the register. Holtek C provides an easy way to access the byte or bit of all registers.

- Byte variable

The syntax of defining a byte variable of the special function register is the same as the data variable with a specific address.

data_type   varaible_name **@** *memory_location*

It is recommended to declare the data_type as 'unsigned char'. For example,

```
unsigned char _a      @ 0x05;
unsigned char _pcl    @ 0x06;
unsigned char _tblp   @ 0x07;
unsigned char _tblh   @ 0x08;
unsigned char _wdts   @ 0x09;
unsigned char _status @ 0x0a;
unsigned char _intc   @ 0x0b;
unsigned char _tmr0h  @ 0x0c;
unsigned char _pa     @ 0x12;
unsigned char _pb     @ 0x14;
```

The usage of the special function registers is the same as that of the ordinary data variables. For example

```
_pa = 0xff; //set PA
if (_pb == (unsigned char)0x80){
...
}
```

- Bit variable

Holtek C compiler provides built-in bit variables for the special function registers. The naming rule of these bit variables is:

_xx_n

xx: the memory address of the function register by two hexadecimal digits.

n: the bit number of the function register

Example

_0a_0 is the bit variable of bit 0 of address 0aH, the carry flag of status register
_12_1 is the bit variable of bit 1 of address 12H, port A

It is not necessary to declare these built-in bit variables before using them, for example, user may assign a meaningful name for each of these bit variables by ***#define*** directive as follow.

// The HT48C50-1

```
#define _c          _0a_0
#define _ac         _0a_1
#define _emi        _0b_0
#define _eei        _0b_1
#define _et0i       _0b_2
#define _et1i       _0b_3
#define _eif        _0b_4
#define _t0f        _0b_5
#define _t1f        _0b_6
#define _pa0        _12_0
#define _pa1        _12_1
#define _pa2        _12_2
```

The data type of these variables is bit. The usage is the same as that of the ordinary bit data variables. For example:

```
bit bflag;
...
_emi = 1;   //enable interrupt
_c = 1;    //set carry
if (_pa0){ //if port A bit 0 set
...
}
bflag = _eei ;
_pa0 = _pa2; //bit assignment
_pa1 = bflag;
```

For each Holtek MCU, there is a corresponding include file which declares the MCU special function registers. The file name of the include file is the same as the MCU name. For example, the HT48C10-1.H is the include file for HT48C10-1 MCU. To access a special function register, either to include the correct MCU include file or to declare the special function register alone.

The following example demonstrates how to access the special function registers. The MCU is HT48C10-1.

```
#include <HT48C10-1.H>

void main(){
   int i;
   _intc = 0;
   _tmrc = 0;
    _tmr = 0;
   _c = 0;      //clear carry flag
   _rrc(&i);   //rotate right through carry
   ...
}
```

Access the IO ports

User can access the Holtek MCU I/O ports by using the same access method of the special function registers. It includes byte variables and bit variables.

Example :

```
unsigned char _pac @ 0x13;
unsigned char _pbc @ 0x15;
#define _pa0 _12_0
#define _pa3 _12_3
#define _pa5 _12_5
#define _pb3 _14_3
#define _pc2 _16_2
#define _pc5 _16_5

void main(){
   _pac = 0xff;    // set port A control register
   _pbc = 0x40;    // set port B control register
   _pa0 = 1;      // set port A bit 0
   _pb3 = 0;      //clear port B bit 3
   _pc5 = _pa3;
   if (_pa5){     //if bit 5 of port A == 1
   ...
   }
   while(! _pc2){  //while bit 2 of port C == 0
   ...
   }
}
```

Built-in Functions

The Holtek C compiler provides some built-in functions which is similar to write assembly instruction directly. Some of these built-in functions are translated to only one assembly instruction. Other built-in functions will facilitate to program with C language.

Assembly-like built-in functions

The following built-in functions will be translated to corresponding assembly instruction by Holtek C compiler.

C Subroutine	Assembly Instruction
void _clrwdt( )	CLR WDT
void _clrwdt1( )	CLR WDT1
void _clrwdt2( )	CLR WDT2
void _halt( )	HALT
void _nop( )	NOP

- Example :

```
//assume the watchdog timer is enabled
//and use one clear WDT instruction

void dotest(){
...
}

void main(){
       unsigned int i;
       for(i=0; i<100; i++){
             _clrwdt();         // CLR WDT
             dotest();
       }
}
```

- Example :

```
//assume the watchdog timer is enabled
//and use two clear WDT instructions

void dotest(){
...
}

void main(){
       unsigned int i;
       for(i=0; i<100; i++){
             _clrwdt1();      // CLR WDT1
             _clrwdt2();      // CLR WDT2
             dotest();
       }
}
```

Rotate functions

There is no rotate operator within the C language, however the Holtek C compiler provides a built-in function for data rotation.

```
void _rr(int*);    //rotate 8 bits data right
void _rrc(int*);   //rotate 8 bits data right through carry
void _lrr(long*);   //rotate 16 bits data right
void _lrrc(long*);  //rotate 16 bits data right through carry
void _rl(int*);    //rotate 8 bits data left
void _rlc(int*);   //rotate 8 bits data left through carry
void _lrl(long*);  //rotate 16 bits data left
void _lrlc(long*); //rotate 16 bits data left through carry
```

For example,

```
#include <HT48C50-1.h>
unsigned int ui;
unsigned long ul;

void error(){
  while(1);
}

void main(){
  ui = 0x1;
  _rr(&ui);    //rotate right
  if (ui != (unsigned int)0x80) error();
  _c = 1;      //set carry
  _rrc(&ui);   //rotate right through carry
  if (ui != (unsigned int)0xc0) error();
  ul = 0xc461;
  _lrl(&ul);   //long rotate left
  if (ul != 0x88c3) error();
  _c = 0;     //clear carry
  _lrlc(&ul); //long rotate left through carry
  if (ul != 0x1186) error();
}
```

Swap function

```
void _swap(int *); //swap nibbles of 8 bit data
```

For example,

```
unsigned int ui;

void error(){
  while(1);
}

void main(){
  ui = 0xab;
  _swap(&ui);
  if (ui != (unsigned int)0xba) error();
}
```

Delay cycle function

```
void _delay(unsigned long)
```

The _delay function forces the MCU to execute the specified cycle count. A value of zero causes an endless loop. The parameter of the _delay could be constant value only. It does not accept a variable.

For example,

```
#define _pa0 _12_0 //port A bit 0
unsigned char _pb @ 0x14 //port B

void error(){
  _delay(0);  //infinite loop. same as while(1);
}

void main(){
  unsigned long time;
  //wrong, parameter should be constant value only
  //_delay(time);

  _pa0 = 1;
  _delay(1); //delay 1 instruction cycle
  _pa0 = 0;
  _delay(15); //delay 15 instruction cycle
  if (_pb != (unsigned int)0x8f) error();
}
```

Programming Tips

Declare variables as unsigned data type

Generally, the operations for unsigned variables are simpler than those for signed variables. It is recommended to declare a variable as an unsigned data type if it does not have a negative value.

Example

```
int i,j;
unsigned int ui, uj;

void test(){
       if (i >= j);      // translate to 8 instructions

       if (ui >= uj);    // translate to 4 instructions
}
```

The first signed comparison is translated into 8 instructions while the second one is translated to only 4 instructions.

Declare variables to be within RAM bank 0

Data located above RAM bank 0 requires indirect accessing which generates some inefficient codes. For those MCUs with multiple RAM banks, it had better to declare the frequently used variables to be within RAM bank 0.

Example

```
//file RAMBANK0.C
//assume the MCU has multiple RAM banks

#pragma rambank0
unsigned int ui0;  // ui0 is in RAM bank 0

#pragma norambank
unsigned int ui;   // ui is relocatable, may not in RAM
                  // bank 0

void test(){
       ui0++;      // translate to 1 instruction
       ui++;       // translate to 5 instructions
}
```

Be care for using the variable declared in RAM bank 0 when a program in other source file wants to access this variable. If a variable is declared to be within RAM bank 0 in file RAMBANK0.C, it could be accessed by programs in other files, ACCESS0.C and ACCESS1.C. But this variable has to be declared as an external and within RAM bank 0 also, otherwise redundant codes or improper codes will be generated. The execution result is unpredictable.

Example

```
// assume the ui0, ui are declared in the above example
// file RAMBANK0.C

// file ACCESS0.C
// declare variables to be the same as RAMBANK0.C

#pragma rambank0
extern unsigned int ui0;   // declare ui0 in RAM bank 0

#pragma norambank
extern unsigned int ui;

void testB(){
       ui0++;  // translate to 1 instruction;  correct
       ui++;    // translate to 5 instructions; correct
}

// file ACCESS1.C
// declare variables to be not the same as RAMBANK0.C

#pragma rambank0
extern unsigned int ui;  // declared ui in rambank0

#pragma norambank
extern unsigned int ui0;

void testC(){
       ui0++;      // 5 instructions; correct
       ui++;       // 1 instruction;  wrong
}
```

In file ACCESS1.C, the ui0++ statement translated into five instructions, four more instructions than the one in the file ACCESS0.C. However this statement is executed correctly. But the ui++ statement is translated to only one instruction and the execution result is unpredictable. The reason is that ui is not defined in RAM bank 0 in file RAMBANK0.C, it should be accessed by the indirect method.

Declare a variable to be Bit type

The bit data variable occupies one bit of memory. If a variable has only two possible values then the bit data type is suitable. Besides the smaller data size to be used, it also generates more compact code.

Example

```
//assume the MCU has single RAM bank
bit bitflag;
unsigned int intflag;
```

```
void test(){
       bitf = 1;          // 1 instruction
       intflag = 1;       // 2 instructions

       if (bitflag);      // 2 instructions
       if (intflag);      // 3 instructions
}
```

Assign an address to a pointer

To assign a constant address to a pointer, the type casting needs to be done explicitly, otherwise the compiler will issue an error.

Example

```
//assume the MCU has multiple RAM banks
int *p1;
unsigned char *p2;
long *p3;

void main(){
       //point to RAM bank 0, offset 0x50
       p1 = (int*)0x50;
       p1 = 0x50;    // error, no casting

       //point to RAM bank 1, offset 0x60
       p2 = (unsigned char *)0x160;

       //point to RAM bank 2, offset 0x30
       p3 = (long *)0x230;
}
```

Get the modulus by more effective method

When you want to get the quotient and the remainder of a division, the following are the most popular statements.

```
q = d1 / d2;
r = d1 % d2;
```

The division subroutine is called by each statement and total is twice. Another effective method to get the quotient and the remainder is to use in-line assembly. The quotient statement is the same, but the remainder is changed to in-line assembly. For 8-bit signed/unsigned division, the remainder will be stored in system variable T3. For 16-bit signed/unsigned division, the remainder will be stored in system variables T4 and T5. T4 is the high byte, T5 is the low byte.

➔ MCU with single RAM bank

- 8 bits division

```
unsigned int d1, d2;
unsigned int q, r;
q = d1 / d2;          // get quotient
#asm
MOV A, T3             ; get remainder
MOV _r,A
#endasm
```

- 16 bits division

```
unsigned long d1, d2;
unsigned long q, r;
q = d1 / d2;
#asm
MOV A, T5
MOV _r,A        ; get low byte remainder
MOV A, T4
MOV _r[1], A    ; get high byte remainder
#endasm
```

➔ **MCU with multiple RAM banks.**

- 8 bits division, r is in ram bank 0

```
unsigned int d1, d2;
unsigned int q;
#pragma rambank 0
unsigned int r;
#pragma norambank

q = d1 / d2;
#asm
MOV A, T3
MOV _r,A
#endasm
```

- 16 bits division, r is in ram bank 0

```
unsigned long d1, d2;
unsigned long q, r;
q = d1 / d2;
#asm
MOV A, T5
MOV _r,A      ;get low byte remainder
MOV A, T4
MOV _r[1], A  ;get high byte remainder
#endasm
```

- 8 bits division, r is not in ram bank 0

```
unsigned int d1, d2;
unsigned int q;
unsigned int r;

q = d1 / d2;
#asm
MOV A, 0E0H
AND [04H],A    ;BP, clear RAM bank and preserve ROM bank
MOV A, BANK _r
OR  [04H], A   ; set bank pointer
MOV A, OFFSET _r
MOV [03H], A   ; move offset to MP1
MOV A, T3
MOV [02H], A   ; move T3 to R1
#endasm
```

- 16 bits division, r is not in ram bank 0

```
unsigned long d1, d2;
unsigned long q, r;
q = d1 / d2;
#asm
```

```
MOV A, 0E0H
AND [04H],A       ;BP, clear RAM bank and preserve ROM bank
MOV A, BANK _r
OR  [04H], A      ; set bank pointer
MOV A, OFFSET _r
MOV [03H], A      ; move offset to MP1
MOV A, T5
MOV [02H], A      ; store to low byte of remainder
INC [03H]         ; point to high byte of remainder
MOV A, T4
MOV [02H], A      ; store to high byte of remainder
#endasm
```

Constant value conversion / casting

The Holtek C compiler is an 8-bit compiler. Note that *int* is equivalent to *char* data type with a range from –128 to +127. If the application operations deals with eight bits constant integers, it is required to cast it into int/char (or unsigned int / unsigned char), otherwise an eight bit hexadecimal integer might be erroneously converted into a 16 bits integer. The constants between 0x80 and 0xff will be converted into the corresponding 16 bit integer without sign extension if no explicit type casting.

Example:

- with explicit type casting, (unsigned int)0xff is equal to an unsigned 8 bits integer with value 255
- with explicit type casting, (int)0xff is equal to a signed 8 bits integer with value –1
- without explicit type casting, 0xff will be implicitly converted into a 16 bit long integer with value 255

Example

```
//assume the MCU has a single RAM/ROM bank
unsigned int ui;
int i;

void main(){
       //8 bit signed comparison
       //5 instructions
       if (i >= 0x7f){
             //equals to if (i >= 127)
       }


       //0x80 implicitly converted to (long)128
       //16 bit signed comparison
       //16 instructions
       if (i >= 0x80){
             // equals to if (i >= 128)
             //always false
       }


       //explicitly casting 0x80 to (int)-128
       //8 bit signed comparison
       //5 instructions
       if (i >= (int)0x80){
             // equals to if (i >= -128)
             //always true
       }
```

```
        //8 bit unsigned comparison
        //4 instructions
        if (ui >= 0x7f){
               // equals to if (ui >= 127);
        }

        //0x80 implicitly converted to (long)128
        //16 bit signed comparison
        //14 instructions
        if (ui >= 0x80){
                //equals to if (ui >= 128L)
        }

        //explicitly casting 0x80 to (unsigned int)128
        //8 bit unsigned comparison
        //4 instructions
        if (ui >= (unsigned int)0x80){
        }
}
```

Serial Port Transmitting Example

This example shows you how to use the Holtek C language to write the time sensitive program. Since the instruction codes translated from the C statements are compiler dependent, the delay constant in the following example might be different under different version compiler. You MUST examine the delay constant when you use it in the first time, update the C compiler or change the MCU. The instructions generated by C compiler are dependent on ROM/RAM single bank or multiple banks.

Preliminary Program

The serial port transmitting protocol is one start bit 0, eight bits data, one stop bit 1. Below is the preliminary program for single RAM bank MCU.

```
// set address 0x12 bit 1 to be output pin (PA1)
#define tx  _12_1

unsigned char sent_val;

void main(){
    _13_1 = 0; //set PA1 as output pin
    sent_val = ‘a’;
    transmit();
}

void transmit(){
    unsigned char sent_bit;
    unsigned char i;

    tx = 0;             // L1 start bit
    for(i=0; i<8; i++){
        sent_bit = sent_val & 0x1;
        sent_val >>= 1;
```

```
            if (sent_bit){
               tx = 1;       // L2
            }
            else {
               tx = 0;       // L3
            }
         }
         tx = 1;             // L4 stop bit
      }
```

The function transmit( ) in above example is not correct due to the transmission baud rate. In order to match the transmission baud rate, a proper delay time should be calculated and inserted before or after each transmitting bit. Because the assembly instructions for output 0 and 1 are different, it has better to use different C statement to output 0 and 1 individually. Hence, it is recommended to replace the statement

```
            tx = sent_bit;
```

with

```
            if (sent_bit){
               tx = 1;      // L2
            }
            else {
               tx = 0;      // L3
            }
```

The statement tx = sent_bit can not determine when it sends 0 or 1.

Adjust Transmitting Timing

Now we need to adjust the timing in order that the instruction cycles between all transmitting (L1 to L2, L1 to L3, L2 to L3, L3 to L2, L2 to L4 or L3 to L4) are the same. After building the program under HT-IDE3000, the Debug window is active.

➔ Adjust L1 to L2 and L1 to L3

- Set Break Points at L1, L2, L3, Open the Cycle Count Window in View menu
- Free Run. ICE will stop at L1
- Modify the sent_val's value to 1 in the Watch Window[1] in Windows menu
- Reset Cycle Count
- Free Run. ICE will stop at L2. cycle count = 0x11
- Reset ICE.
- Free Run. ICE will stop at L1.
- Modify the sent_val's value to 0 in the Watch Window.
- Reset Cycle Count
- Free Run. ICE will stop at L3. cycle count = 0x12

Now, we know the instruction cycles between L1 and L2 is one more than that between L1 and L3. Hence, it should delay one cycle before L2, then both of the instruction cycles from L1 to L2 and from L1 to L3 are all equals to 0x12.

```
            if (sent_bit){
```

[1] In the HT-IDE3000 Watch Window, write dot sent_val (.sent_val) and press Enter. You will see something like ".sent_val :[xxH] = nn". Now you could modify nn to 01. Do not forget to press Enter after your modify otherwise the value will not be modified.

```
                _delay(1);     // add this statement
                tx = 1;        // L2
            }
```

➔ **Adjust L2 to L3 and L3 to L2**

Using the modified code to do below test.

- Set Break Points at L1, L2, L3
- Free Run. ICE will stop at L1.
- Modify the sent_val's value to 5 (00000101b) in the Watch Window.
- Free Run. ICE will stop at L2.
- Reset Cycle Count
- Free Run. ICE will stop at L3. cycle count = 0x12
- Reset Cycle Count
- Free Run. ICE will stop at L2. cycle count = 0x10

Now, the cycle count between L2 and L3 is 0x12, the cycles count between L3 and L2 is 0x10. Hence, it should delay two cycles after L3.

```
            else {
                tx = 0;         // L3
                _delay(2);      // add this statement
            }
```

It is wrong to delay the cycles before L3. Because, it will prolong the period of L1 to L3.

➔ **Adjust L2 to L4 and L3 to L4**

At this moment, the cycle count of the (L1,L2), (L1,L3), (L2,L3), (L3,L2) are all the same. The rest is to check L2 and L4. Using the modified code to do below test.

- Set Break Points at L1, L2,L4
- Free Run. ICE will stop at L1.
- Modify the sent_val's value to 0x80 in the Watch Window.
- Free Run. ICE will stop at L2. (the last loop)
- Reset Cycle Count
- Free Run. ICE will stop at L4. cycle count = 0x8

The delay cycle count should be 10 (0x12-0x8) before L4.

```
        _delay(10);   // add this statement
        tx = 1;       // L4 stop bit
```

Now all the transmission cases have the same period, 18 cycles.

Adjust to Meet the Baud Rate

Baud Rate = SysClk / 4 / (cycle count for transmitting one bit)
Transmit one bit cycle = X+18, X is the additional delay cycle count.
Then the formula of X is

$$X = (\text{SysClk} / \text{Baud Rate} / 4) - 18$$

For example, SysClk = 4MHz and Baud Rate = 9600 then X is equal to 86
The following is the final program.

```
      // This function depends on compiler and MCU.
      // You MUST adjust the delay constants when different
      // compiler or MCU are used

      // suppose address 0x12 bit 1 is the output pin (PA1)
```

```
#define tx   _12_1

unsigned char sent_val;

void transmit(){
   unsigned char sent_bit;
   unsigned char i;

   tx = 0; // L1 start bit
   for(i=0; i<8; i++){
      sent_bit = sent_val & 0x1;
      sent_val >>= 1;
      _delay(86);         // add this statement
      if (sent_bit){
         _delay(1);       // add this statement
         tx = 1;          // L2
      }
      else {
         tx = 0;          // L3
         _delay(2);       // add this statement
      }
   }
   _delay(86+10);         // add this statement
   tx = 1;                // L4 stop bit
   _delay(86);            // add this statement
}
```

The receiving part is similar to the above.

Skeleton Program Example

```
//include files
#include <ht49C50-1.h>

//Interrupt service routines declaration
#pragma vector external_isr @ 0x4
#pragma vector timer0_isr @ 0x8
#pragma vector timer1_isr @ 0xc

//RAM bank undefined variables
unsigned int uia, uib;
unsigned long ula, ulb;

//RAM bank 0 variables
#pragma rambank0
unsigned int uia0, uib0;
unsigned long ula0, ulb0;
bit flag;

//ISR
void external_isr(){
}

void timer0_isr(){
}

void timer1_isr(){
}

//main function
void main(){
}
```

Data Type

Data types

The following table lists the data types, sizes and their range

Data Type	**Size (bits)**	**Range**
bit	1	0,1
char	8	-128~127
unsigned char	8	0~255
short int	8	-128~127
unsigned short int	8	0~255
int	8	-128~127
unsigned	8	0~255
long	16	-32768~32767
unsigned long	16	0~65535

The floating point data type is not supported.

Chapter 4
C Examples

4

In the HT-IDE’s installation path, you could find below examples’ source projects under the <*Sample\C Example*> sub-directory.

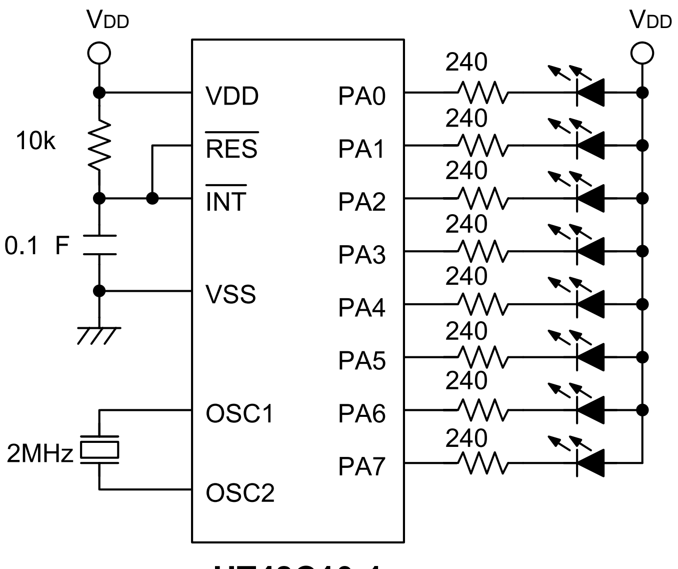
Input/Output Applications

Scanning Light

This example gives a functional emulation of a scanning LED array. Here a row of LEDs will light in turn one after the other. The circuit uses the PA port PA0~PA7, each bit of which is connected via a 240Ω series resistor to an LED.

➔ **Circuit design**

The I/O port bits PA0~PA7 are the outputs, with each output bit controlling a single LED via a 240Ω series resistor. By using the shift right and shift left operator the illuminated LED can be made to move from left to right and vice versa. See the circuit diagram for more details.



HT48C10-1

➔ **Program**

```
//Scan.c
//
//Body: HT48C10-1
//Mask option
//All the mask options use the default value.
```

```
#include <ht48c10-1.h>

bit direction;
unsigned char lamp;

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
    _intc = 0;
    _tmrc = 0;
    _tmr = 0;
    _pac = 0xff; //input mode
    _pbc = 0xff;
    _pcc = 0xff;
}

void main(){

    safeguard_init();

    direction = 0; //shift left direction
    _pac = 0;     //set port A as output port
    lamp = 1;     //set initial lamp light up

    while(1) {
       _pa = lamp;  //output lamp value to port A

       _delay(50000);

       if(!direction)
           lamp <<= 1;
       else
           lamp >>= 1;

       if(lamp & (unsigned char)0x80)
           direction = 1;  //shift right
       else if(lamp & 0x01)
           direction = 0;  //shift left
       //else, don't change the direction
    }
}
```

➔ **Mask option**

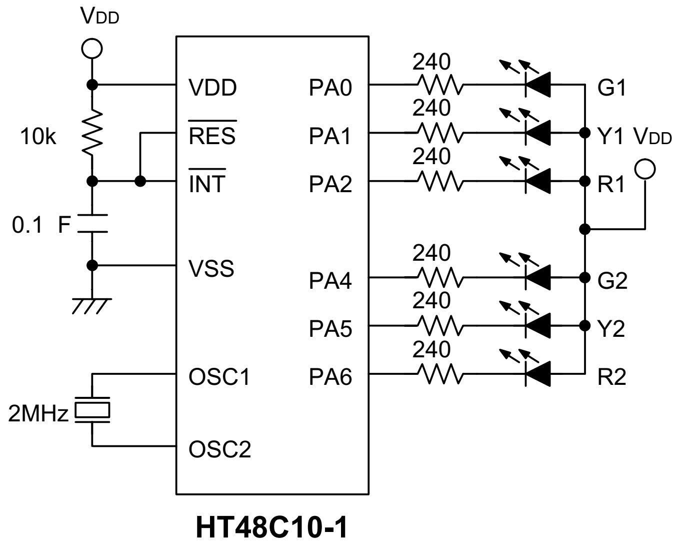
All the mask options use the default value.

Traffic Light

This application uses red, green and yellow LEDs to simulate a crossroads traffic light function. Initially R1 and G2 are illuminated. After a delay the green light flashes followed by the yellow light. After another delay R2 and G1 are illuminated. This cycle will continue in this way indefinitely in the application the different time durations for the red and green light as well as the flashing time can be programmed.

➔ Circuit design

The circuit uses the two port sections PA0~PA2 and PA4~PA6 with each one representing a set of traffic lights on each road at a crossroad intersection. The operation of the circuit will be self explanatory from the contents of the program. See the circuit diagram for more details of the hardware.



➔ Program

```
//Traffic.c
//
//Body: HT48C10-1
//Mask option
//All the mask options use the default value.

#include <ht48c10-1.h>
const unsigned char table[16]={
0x14, 0x4, 0x14, 0x4, 0x14, 0x4, 0x14, 0x24,
0x41, 0x40, 0x41, 0x40,0x41, 0x40, 0x41, 0x42 };

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
    _intc = 0;
    _tmrc = 0;
    _tmr = 0;
    _pac = 0xff; //input mode
    _pbc = 0xff;
    _pcc = 0xff;
}
```

```
//a long time delay
void mydelay(unsigned int times){
    while(times--) _delay(65000);
}

void main(){
    unsigned char i, j, idx;

    safeguard_init();

    _pac = 0; //set port A to output port
    _pa = 0; //zero port A (all light on)
    while(1) {
       idx = 0;
       for(i=0; i!=2; i++) {
           _pa = table[idx];
           idx++;
           mydelay(8);
           for(j=0; j!=6; j++) {
                 _pa = table[idx];
                 idx++;
                 mydelay(1);
           }
           _pa = table[idx];
           idx++;
           mydelay(4);
       }
    }
}
```

➔ Mask option

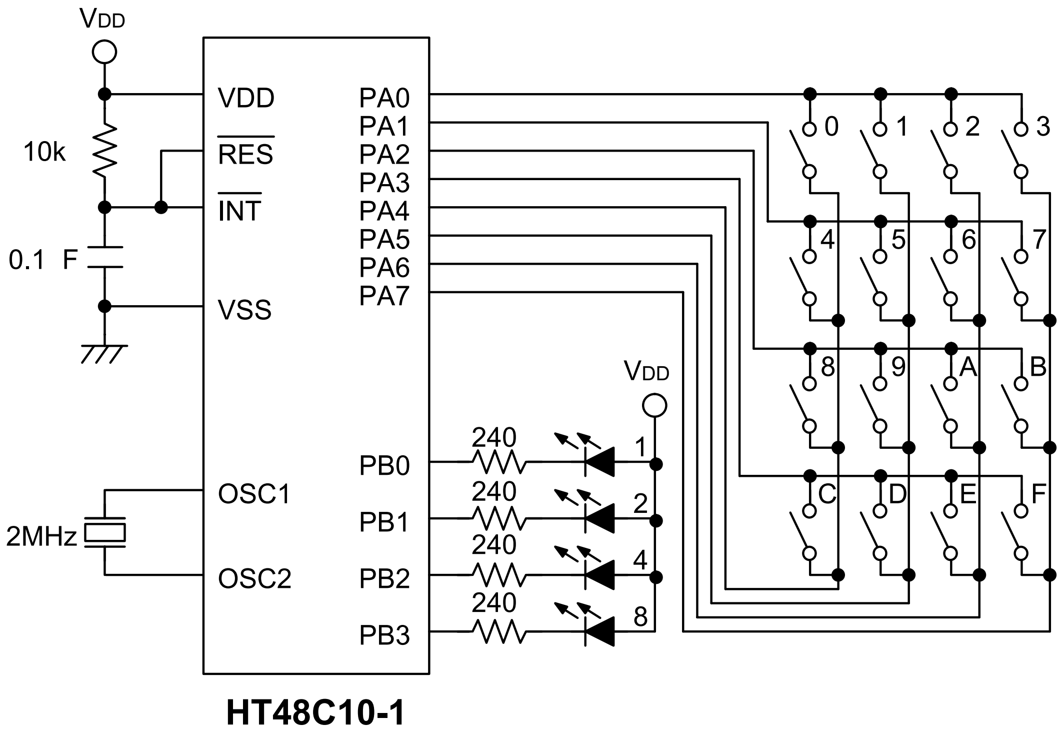
All the mask options use the default value.

Keyboard Scanner

This application uses a 4x4 keyboard matrix, giving a total of 16 keys with each key representing a single hexadecimal value as shown in the diagram. The program scans the keyboard matrix to detect which key was pressed and after detection displays on the LED display the corresponding hex code. There are 4 LEDs, so a range of values from 0000 to 1111 can be displayed. During the scanning process, if two keys are pressed simultaneously only the first key scanned will be detected and displayed. By using this method 8 logic lines can control up to 16 switches with required values assigned to each key.

➔ Circuit design

PA0~PA3 are assigned as outputs and PA4~PA7 assigned as inputs, together forming a 4x4 matrix. Note that during creation of the project, PA/PA should have the pull-high option selected from the mask option and the BZ/BZB should select "All Disable". The program detects which key was pressed while a look up table defines the value of each key. PB0~PB3 are defined as outputs and represent a 4 bit hex code giving 16 different values with each value representing a single key.



➔ **Program**

```
//Keyboard.c
//
//Body: HT48C10-1
//Mask option
//BZ/BZB : All Disable
//the others use the default value

#include <ht48c10-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
    _intc = 0;
    _tmrc = 0;
    _tmr = 0;
    _pac = 0xff; //input mode
    _pbc = 0xff;
    _pcc = 0xff;
}

const unsigned char led_code[16]=
    {0xff, 0xfe, 0xfd, 0xfc, 0xfb, 0xfa, 0xf9, 0xf8,
     0xf7, 0xf6, 0xf5, 0xf4, 0xf3, 0xf2, 0xf1, 0xf0};
const unsigned char scan[4] = {0xfe, 0xfd, 0xfb, 0xf7};
```

```
//return the row number of the pressed key
unsigned char wait_key_pressed(){
    unsigned char i;
    i=0;
    while(1){
       _pac = scan[i];  //output scan code to port A
       if ((~_pa) & (unsigned char)0xf0){ //key pressed
           _delay(2000); //debounce
           //after debounce, if the key is still pressed
           //we claim it a key pressed, otherwise ignore it
           if ((~_pa) & (unsigned char)0xf0)
                 return i; //row i, key pressed
       }
       i++;
       if (i > 3) i = 0;
    }
}

// return the column number of the pressed key
unsigned char wait_key_released(){
    unsigned char i;
    unsigned char key;

    key = _pa; //keep the pressed key

    // wait until key released
    while((~_pa) & (unsigned char)0xf0);

    //find out which column key pressed
    //no debouce needed
    for(i=0; i<4; i++)
       if ((~key) & (0x10<<i))
           break; //column i, key pressed
    return i;
}

unsigned char get_key(){
    unsigned char row, col;
    row = wait_key_pressed();
    col = wait_key_released();
    return (row << 2) + col;
}

void main(){
    unsigned char index;

    safeguard_init();

    _pac = 0xff; //set port A as input port
    _pbc = 0x00; //set port B as output port
    _pa = 0;    //zero port A
    _pb = 0xff;  //off LEDs

    while(1){
       index = get_key();
       //the key value won't be displayed until
       // the key is released
       _pb = led_code[index];
    }
}
```

➔ **Mask option**

The BZ/BZB mask option selects *All Disable*, the others use the default value.

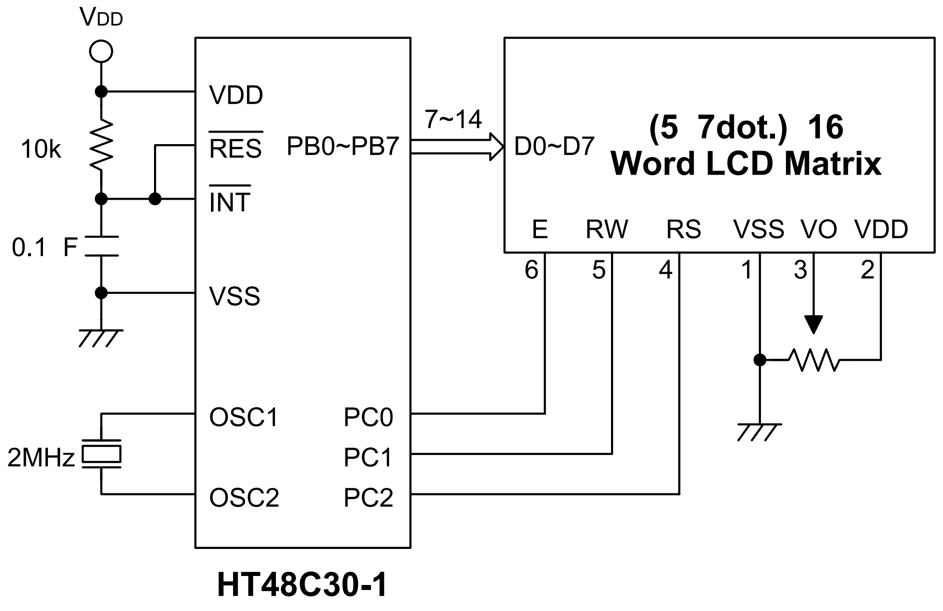
LCM

This application describes the use of an 8-bit microcontroller used in conjunction with a DV16100NRB liquid crystal display. This LCM is driven and controlled by an internal Hitachi HD44780 device. In this application only the timing requirements of the LCM need to be considered to produce the correct microcontroller signals. for more detailed timing and instruction information, the LCM manufacturer's data should be consulted first.

LCMs can operate in either 4 bit or 8 bit mode. Using a 4 bit mode of operation, transmitting a character or an instruction to the module requires two transmission events to complete the operation. With an 8-bit mode of operation only one transmit event is required, however an extra 4 I/O lines are required.

➔ Circuit design

PB0~PB7 are setup as I/O bits while PC0~PC2 as the LCM control lines are setup as outputs. These can be setup according to the specific user requirements.



➔ Program

```
//Lcm.c
//
//Body: HT48C30-1
//Mask option
//BZ/BZB : All Disable
//the others use the default value

#include <ht48c30-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
```

```
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmrc = 0;
       _tmr = 0;
       _pac = 0xff; //input mode
       _pbc = 0xff;
       _pcc = 0xff;
}

//#define FOUR_BIT
//#define ONE_LINE

//for DV-16100NRB
// port B : LCM data port
// port C : LCM control port
#define LCM_CLS       0x1
#define CURSOR_HOME    0x2
#define CURSOR_SR     0x14
#define CURSOR_SL     0x10
#define INCDD_CG_SHF_C  0x6
#define TURN_ON_DISP   0xf
#define LCD_ON_CSR_OFF  0xc

#define LCM_DATA      _pb
#define LCM_DATA_CTRL   _pbc
#define LCM_CTRL      _pc
#define LCM_CTRL_CTRL   _pcc

#define LCM_CTRL_E     _pc0
#define LCM_CTRL_RW    _pc1
#define LCM_CTRL_RS    _pc2

#define WRITE(a)      { \
LCM_DATA = (a); \
LCM_CTRL_E = 1; \
LCM_CTRL_E = 0;}

const unsigned char msg[16] = "HOLTEK 8 bit MCU";

void mydelay(unsigned char ct);
void LCM_initialize();
void send_cmd(unsigned char);
void write_char(unsigned char);
void busy_check(void);
void lcm_delay(void);

void main(){
       unsigned int i;

       safeguard_init();
       LCM_initialize();

       while(1){
             send_cmd(LCM_CLS);
             mydelay(2);
             send_cmd(CURSOR_HOME);
             for(i = 0; i<sizeof(msg); i++){
                   if (i == 8)
                         send_cmd(0xc0); //move cursor
                                        //to 2nd line
```

```
                    write_char(msg[i]);   //(1st line:00h~,
                                           // 2nd line:40h~)
                }
                send_cmd(LCD_ON_CSR_OFF);
                mydelay(5);
        }
}

void mydelay(unsigned char ct){
        while(ct--) _delay(65535);
}

void LCM_initialize(){
        LCM_DATA_CTRL = 0;//setup LCM data port as output port
        LCM_CTRL_CTRL = 0;//setup LCM control port as output port
        LCM_DATA = 0;    //clear LCM data port
        LCM_CTRL = 0;    //clear LCM control port

#ifdef FOUR_BIT
        WRITE(0x20); //4 bit mode
#else
        WRITE(0x30); //8 bit mode
#endif

//According to the data for the HD44780, there needs to be at
//least 4.5 ms delay between each program.
        mydelay(1);

#ifdef FOUR_BIT
#ifdef ONE_LINE
        WRITE(0x20); //4-bit 1-line
#else
        WRITE(0x28); //4-bit 2-line
#endif
#else
#ifdef ONE_LINE
        WRITE(0x30); //8-bit 1-line
#else
        WRITE(0x38); //8-bit 2-line
#endif
#endif

#ifdef FOUR_BIT
        WRITE(0x80); //4-bit high nibble (2nd pass)
#endif
        send_cmd(LCM_CLS);          //clean display
        send_cmd(TURN_ON_DISP);     //turn on display
        send_cmd(INCDD_CG_SHF_C);   //auto increment mode
                              //cursor left and DD RAM address+1
}


// send command to LCM
void send_cmd(unsigned char c){
#ifdef FOUR_BIT
        unsigned char tmp;
        tmp = c << 4;
        c &= (unsigned char)0xf0;
#endif
        busy_check();
        LCM_DATA = c;
        LCM_CTRL_RW = 0;
        LCM_CTRL_RS = 0;
        LCM_CTRL_E = 1;
        LCM_CTRL_E = 0;
#ifdef FOUR_BIT
        WRITE(tmp);
#endif
}

// write character to LCM
void write_char(unsigned char c){
#ifdef FOUR_BIT
        unsigned char tmp;
        tmp = c<<4;
        c &= (unsigned char)0xf0;
#endif
        busy_check();
        LCM_DATA = c;
        LCM_CTRL_RW = 0;
        LCM_CTRL_RS = 1;
        LCM_CTRL_E = 1;
        LCM_CTRL_E = 0;
#ifdef FOUR_BIT
        WRITE(tmp);
#endif
}

// Wait until the busy flag is not busy
void busy_check(void){
        unsigned char val, tmp;
        do{
                LCM_CTRL_E = 0;
                LCM_DATA_CTRL = 0xff;
                LCM_CTRL_RS = 0;
                LCM_CTRL_RW = 1;
                LCM_CTRL_E = 1;
                val = LCM_DATA;
                LCM_CTRL_E = 0;
#ifdef FOUR_BIT
                tmp = val & (unsigned char)0xf0;//4-bit high nibble
                LCM_CTRL_E = 1;   //pulse high
                val = LCM_DATA;    //4-bit low nibble (2nd pass)
                LCM_CTRL_E = 0;   //pulse low
                val = (val>>4) | tmp; //combine 2 pass
#endif
        }while(val & (unsigned char)0x80);
        LCM_CTRL_RW = 0;
        LCM_DATA_CTRL = 0;   //LCM not busy, then set LCM data
                             //bus to input port
}
```

➔ **Mask option**

The BZ/BZB mask option selects *All Disable*, the others use the default value.

Using an I/O Port as a Serial Application

This application shows code to simulate serial port operation. This can be used as a basis for the development of simple serial port applications such as 8-bit communication, non-parity, single stop bit applications.

➔ **Program**

```
//Serial.c
//
```

```
//Body: HT48C70-1
//Mask option
//WDT : Disable
//the others use the default value

#include <ht48c70-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0
void isr_c(){} // timer/event 1

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmr0c = 0;
       _tmr0h = 0;
       _tmr0l = 0;
       _tmr1c = 0;
       _tmr1h = 0;
       _tmr1l = 0;
       _pac = 0xff;
       _pbc = 0xff;
       _pcc = 0xff;
       _pdc = 0xff;
       _pec = 0xff;
       _pfc = 0xff;
       _pgc = 0xff;
}

#define tx   _pa3            //transmit pin
#define rx   _pa2            //receive pin
#define _pac3 _13_3
#define _pac2 _13_2

unsigned char data;

void transmit(unsigned char);
void receive(unsigned char *);


//system frequency: 4MHz
//#define T 38 //baudrate 19200 = 4M/4/(T+14) => T = 38
#define T 90 //baudrate 9600 = 4M/4/(T+14) => T = 90
//#define T 194 //baudrate 4800 = 4M/4/(T+14) => T = 194
//#define T 402 //baudrate 2400 = 4M/4/(T+14) => T = 402

void main(){
       safeguard_init();

       _pac2 = 1;     //set receive pin to input mode
       _pac3 = 0;     //set transmit pin to output mode

       while(1){
              receive(&data);
              transmit(data);
       }
}

void transmit(unsigned char val){
        unsigned char i;

        tx = 0;
        for(i=0; i<8; i++){
              _delay(T);
              if (val & 1) tx = 1;
              else tx = 0;
              val >>= 1;
        }
        _delay(T);
        tx = 1;
        _delay(T);
}

void receive(unsigned char *val){
        unsigned char i, v;

        v = 0;
        while(rx); //wait start bit
        for(i=0; i<8; i++){
              _delay(T);
              if (rx) v |= (unsigned char)0x80;
              v >>= 1;
        }
        _delay(T);//skip stop bit
        _delay(T);
        *val = v;
}
```

➔ Mask option

The WDT mask option selects *Disable*, the others use the default value.

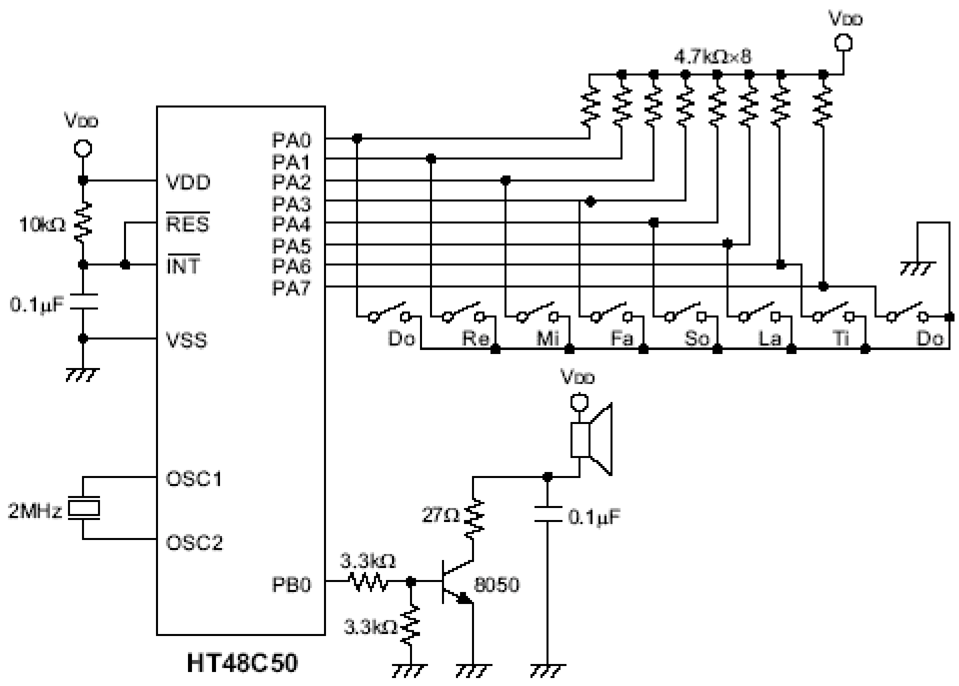
Interrupt and Timer/Counter Applications

Electric Piano

This example describes how to implement a scanning keyboard and then from the pressed key generate a corresponding defined sound frequency. Each time a key is pressed the corresponding frequency value is placed into the timer/counter register. When this counter counts to its maximum value an internal interrupt is generated and the interrupt routine is run. At this point the timer/counter register value is reloaded and the counting continues. In this way, by programming different values into the timer/counter register, different values of frequency can be generated. The internal interrupt routine contains code to change the state of the output port and thus generate the required frequency on a corresponding pin and create the desired note. By adding a suitable amplifier and speaker the system is complete. The important point of the software is to use the timer/counter as a counter to control the output frequency. This frequency has to be calculated.

➔ Circuit design

The I/O port bits PA0~PA7 are the outputs, with each output bit controlling a single LED via a 240Ω series resistor. By using the shift right and shift left operator the illuminated LED can be made to move from left to right and vice versa. See the circuit diagram for more details.



➔ **Program**

```
//Piano.c
//
//Body: HT48C50-1
//Mask option
//BZ/BZB : All Disable
//the others use the default value

#include <ht48c50-1.h>

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmr0c = 0;
       _tmr0 = 0;
       _tmr1c = 0;
       _tmr1h = 0;
       _tmr1l = 0;
       _pac = 0xff;
       _pbc = 0xff;
       _pcc = 0xff;
       _pdc = 0xff;
}


#define _tmr1c4 _11_4  //timer1 enable bit
```

```
const unsigned char frq[16] = {
0x21, 0xfe, 0x58, 0xfe, 0x84, 0xfe, 0x99, 0xfe,
0xc1, 0xfe, 0xe3, 0xfe, 0x2, 0xff, 0x11, 0xff};

unsigned char frq_idx;

void initial();
void wait_key_press();
void wait_key_release();
void start_sound();
void stop_sound();

void main(){

       safeguard_init();
       initial();

       while(1){
              wait_key_press();
              start_sound();
              wait_key_release();
              stop_sound();
       }
}

void wait_key_press(){
       unsigned char i, key;

       key = 0;
       while(!key)
              key = ~_pa;

       for(i=0; i<8; i++){
              if (key & 0x1){
                     frq idx = i << 1;
                     break;
              }
              key >>= 1;
       }
}

void wait_key_release(){
       unsigned char key;
       key = 1;
       while(key)
              key = ~_pa;
}

void start_sound(){
       _intc = 9;             //enable timer1
       _tmr1c = 0x80;          //timer mode
       _tmr1l = frq[frq_idx];  //load sound freq.
       _tmr1h = frq[frq_idx+1];
       _tmr1c4 = 1;           //start timer1
}

void stop_sound(){
       _tmr1c4 = 0;           //stop timer1
       _pb = 0;
}

void isr_c(){                  // timer1
       _pb = ~_pb;            // generate square wave
}
```

```
void initial(){
      _pac = 0xff;    //set port A to input port
      _pbc = 0;      //set port B to output port
      _pb = 0;
}
```

➔ Mask option

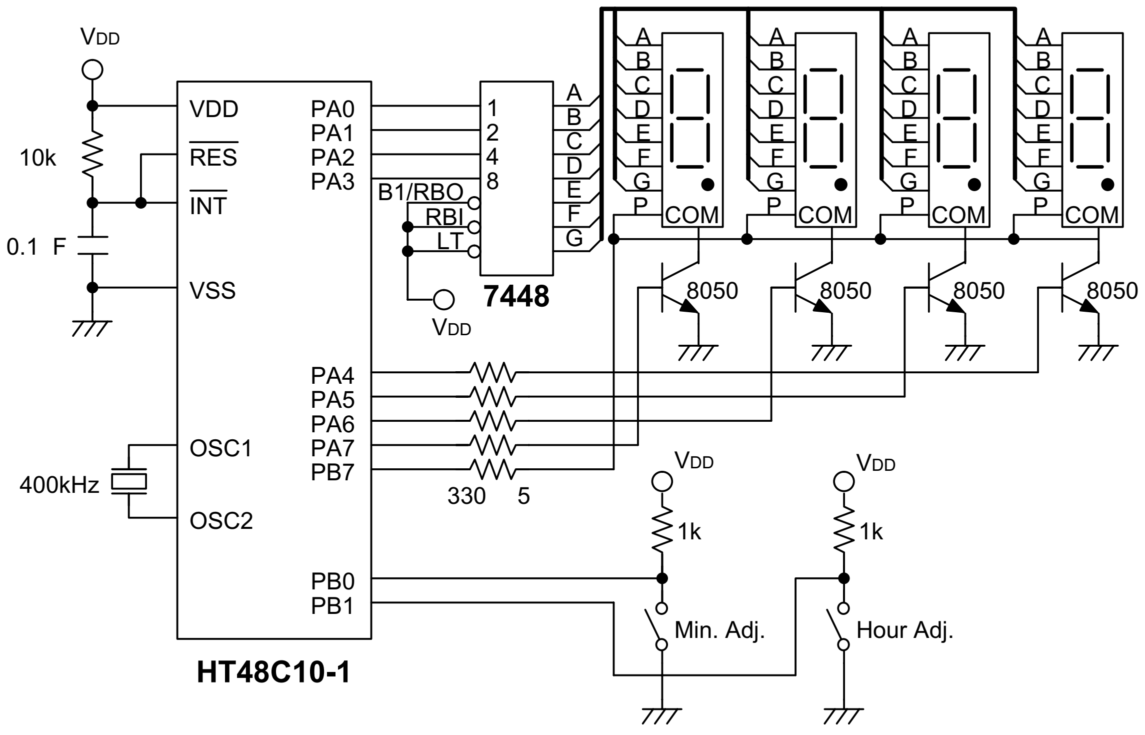
The BZ/BZB mask option selects *All Disable*, the others use the default value.

Clock

This application shows the use of the 16 bits of the timer counter to generate internal interrupts and consequently generate a timing function. This application depends upon the system clock frequency as a basis for its timing. The application shown here uses a 400KHz system clock which will generate a 100KHz timer/counter clock due to the internal divide by four operation. With a 16 bit counter the maximum count is 65536, this would generate an internal interrupt every 0.65536 seconds. However for a clock function a basic time unit of 1 second is required so for this reason the timer/counter is setup to record a basic timing of 0.5 seconds. In this case an interrupt will be generated every 0.5 seconds, so by counting two interrupts a means of obtaining the basic timing unit of 1 second is obtained. The application shown uses a 4 seven segment displays to display a clock in 24 hour format, displaying both hours and minutes. Two keys are provided to provide for adjustment of hours and minutes.

➔ Circuit design

PA0~PA7 are setup as outputs with PA0~PA3 setup as the display data. PA4~PA7 provide scanning inputs to the control transistors for the segment displays. These will scan the individual displays one after the other. PB0 and PB1 are setup as inputs for the switches which enable the hours and minutes to be preset.



➔ **Program**

```
//Clock.c
//
//Body: HT48C50-1
//Mask option
//BZ/BZB : All Disable
//SysFreq: 400KHz
//the others use the default value

#include <ht48c50-1.h>

#define _tmr1c4 _11_4  //timer4 enable bit
#define min_adj_button  _pb0
#define hour_adj_button _pb1

#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc

//ISR for safequard
void isr_4(){} // external ISR
void isr_8(){} // timer/event 0

//initialize registers for safeguard
void safeguard_init(){
       _intc = 0;
       _tmr0c = 0;
       _tmr0 = 0;
       _tmr1c = 0;
       _tmr1h = 0;
       _tmr1l = 0;
       _pac = 0xff;
       _pbc = 0xff;
       _pcc = 0xff;
       _pdc = 0xff;
}

void initial();
void check_time();
void show_clock();
unsigned char min_adj_pressed();
unsigned char hour_adj_pressed();
void min_adjust();
void hour_adjust();
void arrange_hour();
void set_timer();

unsigned char half_second;
unsigned char min_l, min_h;
unsigned char hour_l, hour_h;

void main(){

       safeguard_init();
       initial();

       while(1){
              check_time();
              show_clock();
              if (min_adj_pressed())  min_adjust();
              if (hour_adj_pressed()) hour_adjust();
       }
}
```

```
void isr_c(){          //timer1
      half_second++;
      _pb = ~_pb;    //flash 'dot' every 0.5 second
}

void initial(){
      _pac = 0;      //set port A to output port
      _pbc = 0x7f;   //set port B to input port exclude pb7
      _pb = 0;
      _pa = 0;

      min_l = 0;
      min_h = 0;
      hour_l = 0;
      hour_h = 0;
      half_second = 0;

      _intc = 0x9;   //enable timer1
      _tmr1c = 0x80;  //timer1 mode (internal clock)
      set_timer();
}

//check if the min_adj_button is pressed or not
//return 1: if the min_adj_button is pressed
//       0: otherwise
unsigned char min_adj_pressed(){
      if (min_adj_button == 0){//pressed
            _delay(2000); //debounce
            if (min_adj_button == 0)
                  return 1; //still pressed, recognize it
      }
      return 0;
}

//check if the hour_adj_button is pressed or not
//return 1: if the hour_adj_button is pressed
//       0: otherwise
unsigned char hour_adj_pressed(){
      if (hour_adj_button == 0){//pressed
            _delay(2000); //debounce
            if (hour_adj_button == 0)
                  return 1; //still pressed, recognize it
      }
      return 0;
}

void check_time(){
      if (half_second >= 120){
            half_second -= 120;
            min_l++;
            if (min_l >= 10){
                  min_l = 0;
                  min_h++;
                  if (min_h >= 6){
                        min_h = 0;
                        hour_l++;
                        arrange_hour();
                  }
            }
      }
}

//This function is to arrange the hour value
void arrange_hour(){
        if (hour_h == 2 && hour_l == 4){
                hour_h = 0;
                hour_l = 0;
        }
        else if (hour_l == 10){
                hour_l = 0;
                hour_h++;
        }
}

void show_clock(){
        _pa = min_l | 0x10;
        _pa = min_h | 0x20;
        _pa = hour_l | 0x40;
        _pa = hour_h | 0x80;
}

//This function is to adjust the minute.
//The minute will increase 1 when the min_adj_button is pressed .
//If the button is held longer than 1.5 seconds, the minute will
//increase 1 every 0.5 second
void min_adjust(){
        bit held_long_time = 0;

repeat_inc:
        min_l++;
        if (min_l >= 10){
                min_l = 0;
                min_h++;
                if (min_h >= 6) //don't care hour
                        min_h = 0;
        }
        half_second = 0;
        while(min_adj_button == 0){//while min_adj_button
                show_clock();           // is held
                if (!held_long_time){
                        if (half_second>2){//longer than 1.5 sec
                                held_long_time = 1; //set flag
                                goto repeat_inc; //increase minute
                        }
                        //less than 1.5 seconds, do nothing
                }
                else{
                        if (half_second)
                                goto repeat_inc; //inc 1, 0.5 sec
                        //less than 0.5 second, do nothing
                }
        }
        half_second = 0;
        set_timer();
}

//This function is to adjust the hour.
//The hour will increase 1 when the hour_adj_button is pressed .
//If the button is held longer than 1.5 seconds, the hour will
//increase 1 every 0.5 second
void hour_adjust(){
        bit held_long_time = 0;

repeat_inc:
        hour_l++;
        arrange_hour();
        half_second = 0;
        while(hour_adj_button == 0){
                show_clock();
```

```
            if (!held_long_time){
                  if (half_second>2){//longer than 1.5 sec
                        held_long_time = 1; //set flag
                        goto repeat_inc;//increase hour
                  }
                  //less than 1.5 seconds, do nothing
            }
            else{
                  if (half_second)
                        goto repeat_inc;//inc 1, 0.5 sec
                  //less than 0.5 second, do nothing
            }
      }
      half_second = 0;
      set_timer();
}

void set_timer(){
      _tmr1c4 = 0;
      _tmr1l = 0xb0;
      _tmr1h = 0x3c;
      _tmr1c4 = 1;   //start timer1
}
```