

## Indirect Branch Destination Register Corruption on the UT80CXX196KD

Table 1: Cross Reference of Affected Product Revisions

Product Name:	SMD #:	Device Type:	Internal PIC Number:
UT80CRH196KD	5962R98583	01 and 02	JD02A through JD02D
UT80C196KD	5962-98583	01 and 02	JD02A through JD02D

### 1.0 Overview

This product erratum is provided by Aeroflex UTMC to identify a design flaw associated with the implementation of the indirect branch instruction ( syntax: BR [wreg] ) on the UT80CXX196KD. This erratum is directed to all software engineers who are developing software applications for the UT80CXX196KD. This erratum describes the behavior of the indirect branch instruction, C-Code instructions that can generate an indirect branch instruction, and offers possible work-arounds to the indirect branch flaw.

### 2.0 BR [wreg] Functional Discussion

The branch instruction is used to continue program execution from a new location in program memory. The syntax for the indirect branch instruction is BR [wreg]. The *wreg* is defined as a word register located in the internal register file (address 00H through FFH) of the UT80CXX196KD. The contents of the *wreg* represent the absolute program memory address where the UT80CXX196KD shall continue execution following the branch instruction.

Proper execution of the indirect branch instruction results in the UT80CXX196KD moving the contents of the *wreg* into the program counter (PC) with the next instruction fetch coming from the new address defined in the PC. Correct execution of the indirect branch instruction makes no mention of modifying the *wreg* contents. Therefore, subsequent execution of that same branch instruction should always continue execution from the same address unless a separate instruction intentionally changes the contents of the *wreg*.

The UT80CXX196KD implementation of the indirect branch instruction satisfies the expected program flow control requirements defined in the MCS-96 instruction set architecture, but it also modifies the contents of the *wreg* after the original contents are copied into the program counter. If the same indirect branch instruction is executed without an intervening update of the *wreg*, the UT80CXX196KD branches to an undesired program memory location. If your application software is written in assembly language you can easily ensure that the *wreg* used in an indirect branch instruction is updated prior to the indirect branch. C-code, on the other hand, is not as straight forward to predict or control. The remaining section of this erratum offers a several C-code examples that compile to an indirect branch instruction and recommends a possible work-around to each corresponding code segment.

### 3.0 Avoiding the Indirect Branch Failure

Based on its research, Aeroflex UTMC has identified two possible C constructs that, when compiled, generate the indirect branch instruction. The first construct is the *switch* statement which uses an indirect branch instruction to continue processing from the *case* label that matches the input argument. The second, and more critical, construct uses an indirect branch as the pointer to a function. The basic operation of the pointer to a function is that the pointer behaves as the *wreg* portion of an indirect branch statement. It is important to note that Aeroflex UTMC does not guarantee that this is an all inclusive list of C constructs that are capable of generating an indirect branch instruction.

### 3.0.0 Indirect Branch Generated by the “SWITCH” Statement

Although the *switch* statement (reference the following sample program) uses an indirect branch to continue program execution from a desired *case* label, it always initializes the *wreg* prior to executing the indirect branch. As a result, Aeroflex UTMC does not foresee a way for the *switch* statement to be affected by the indirect branch flaw.

```
//C-Code that generates an indirect branch instruction through a SWITCH statement.
//Compiled using Tasking EDE

void main(void)
{
    volatile int ret;

    // The switch statement below will create a br [tmp0]
    // instruction. 6 cases plus the default were
    // needed to create this instruction. Fewer
    // statements do not generate the indirect branch.

    while( 1 )
    {
        switch(ret)
        {
            case 0:
                ret++;
                break;
            case 1:
                ret++;
                break;
            case 2:
                ret++;
                break;
            case 3:
                ret++;
                break;
            case 4:
                ret++;
                break;
            case 5:
                ret++;
                break;
            default:
                ret = 0;
                break;
        } //end switch

    } //end while

} //end main
```

## 3.0.1 Indirect Branch Generated by a Pointer to a Function

Unlike the *switch* statement, a pointer to a function causes a program fault unless your software takes appropriate steps to ensure the pointer is located outside of the UT80CXX196KD register space or is updated before the indirect branch executes. The following software example demonstrates how you could use a pointer to a function. If you ran this program on the UT80CXX196KD, it would fail on the second pass through the while loop.

```
//C-Code that generates and indirect branch through a pointer to a function.
//Compiled using Tasking EDE

//   Simple counter to demonstrate using a pointer to a function:

unsigned char loop_inc ( unsigned char loop_count )
{
    return ++loop_count;
} //end loop_inc

void main(void)
{
    unsigned char loop = 0;

    //Define a pointer to a function

    unsigned char (*pf) (unsigned char);

    //Assign address of loop_inc() to pf

    pf = loop_inc;

    while( loop <= 100 )
    {

        //Call the function pointed to by pf. This function call will result in an
        //indirect branch which will fail on the second pass through the while loop.
        //The resulting indirect branch syntax: br [pf]

        loop = pf ( loop );

    } //end while

    for ( ; ; ) //Loop forever

} //end main
```

Although you may ask why would you want to use such an unusual technique to call a function, it is a very powerful tool in C. Specifically, pointers to functions provide an elegant way to call different functions based on the input data. In the example above, the C instruction: `loop = pf ( loop )` is compiled to a sequence of assembly instructions that include the indirect branch `br [pf]`. Because this program only assigns `pf` to the function `loop_inc` one time, it stands to reason that the indirect branch expects the contents of `pf` to be constant the entire time the program executes within the `while` loop.

Because the UT80CXX196KD modifies the contents of `pf` after its contents are copied to the program counter, the second pass through the `while` loop results in a failure. When the `br [pf]` instruction executes during the second pass through the loop, the corrupted contents of `pf` will be loaded into the program counter and the UT80CXX196KD continues operation at an unknown instruction location.

Fortunately, there are two methods to protect your pointer to a function from failing the indirect branch instruction. The first solution is to assign the pointer to the desired function immediately prior to making the function call. This ensures the pointer contents are updated with the correct value before they are used in the subsequent indirect branch instruction.

The second solution is to force the linker to place the pointer in a memory location residing outside of the register space defined in the UT80CXX196KD. This can be performed using the `#pragma locate( var=addr )` construct offered by the Tasking EDE. The `#pragma locate( var=addr )` construct allows you to place a regular expression in the parenthesis to define the memory location where the variable on the left side of the expression links to the address defined by the result of the expression on the right side of the equal sign. Please refer to the following sample program to see how each of these solutions can be implemented to prevent the indirect branch flaw from affecting the pointer to a function construct.

```
//C-Code that demonstrates indirect branch protection for the pointer to a function construct
//Compiled using Tasking EDE.
```

```
// Define and locate the pointer to an even address above 0xFE.
unsigned char (*pf) (unsigned char);
#pragma locate(pf=0x100)
```

```
// Simple counter to demonstrate using a pointer to a function
unsigned char loop_inc ( unsigned char loop_count )
{
    return ++loop_count;
} //end loop_inc
```

```
void main(void)
{
```

```
    unsigned char loop = 0;
    unsigned char decade = 0;
```

```
    //Define a pointer to a function
    unsigned char (*pdecade) (unsigned char);
```

```
    //Assign address of loop_inc() to pf
    pf = loop_inc;
```

```
while( decade < 50 )
{
    //Call the function pointed to by pf. This function call will result in an
    //indirect branch which will not fail because the contents of pf will first
    //be copied to a temporary register every time the C instruction executes.
    //Resulting indirect branch syntax: BR [TMP0]

    loop = pf ( loop );

    if (loop == 10)
    {
        //Assign address of loop_inc() to pdecade
        pdecade = loop_inc;

        //Call the function pointed to by pdecade. This function call will pass
        // because pdecade is always initialized before the indirect branch executes.
        //The resulting indirect branch syntax: BR [pdecade]
        decade = pdecade ( decade );

    } //end if

    //The while loop should exit with decade = 50 and loop = 10.

} //end while

for ( ; ; ); //Loop forever

} //end main
```

### **3.1 Avoiding the Indirect Branch Flaw**

Because the UT80CXX196KD modifies the contents of the *wreg* after it copies the original contents to the program counter during an indirect branch instruction, software that uses this instruction should ensure that the contents of the *wreg* are either updated prior to the indirect branch instruction or protected so they are never lost following the execution of the indirect branch. Assembly language based programs have complete control over the usage of the indirect branch and should therefore have no trouble ensuring that the *wreg* is always accurate prior to the execution of the indirect branch. C based software applications on the other hand are not as strait forward to control and, therefore, requires more knowledge and caution on the part of the software engineer to ensure all compiled indirect branch instructions are protected.

Concerning the C based programs supplied in this erratum to demonstrate the failure mechanisms and associated work-arounds, Aeroflex UTMC provides a Tasking EDE project that includes all of the “at-risk” instructions defined in this erratum and implements the recommended work-arounds. The software is compiled using the Tasking EDE and designed to run on an Intel 80C196KX Evaluation Board. You can download the zipped project file by clicking on the following link:

Example Software Project: [www.utmc.com/products/indir\\_branch.zip](http://www.utmc.com/products/indir_branch.zip)

If the software workarounds described in this erratum are insufficient for your application, the only remaining solution is to use the latest member of the UT80CXX196KD family -- the UT80CRH196KDS (Aeroflex UTMC internal PIC# KC01A). Aeroflex UTMC is currently accepting orders for the UT80CRH196KDS. For more information on the UT80CRH196KDS please download the datasheet or UT80CXX196KD and UT80CRH196KDS differences list by clicking on one of the following links:

UT80CRH196KDS Datasheet: [www.utmc.com/products/ut80196s.pdf](http://www.utmc.com/products/ut80196s.pdf)

UT80CXX196KD and UT80CRH196KDS Differences: [www.utmc.com/products/ut80196\\_kd\\_kds\\_diff.pdf](http://www.utmc.com/products/ut80196_kd_kds_diff.pdf)