マイクロコントローラ技術情報

<table>
<tr><td rowspan="3">技術通知

V850　C コンパイラ・パッケージ
CA850
使用制限事項の件</td><td>発行番号</td><td>ＺＢＧ－ＣＤ－１０－００３０号</td><td>1/3</td></tr>
<tr><td>発行日</td><td colspan="2">２０１０年　９月　１３日</td></tr>
<tr><td>発行部門</td><td colspan="2">ルネサス エレクトロニクス株式会社
ＭＣＵ事業本部
ソフトウェア統括部
ＭＣＵツール技術部</td></tr>
</table>

文書分類	○	使用制限事項		バージョン・アップ		ドキュメント誤記訂正（正誤表）		その他

関連資料	CA850 Ver.3.20 C コンパイラ・パッケージ C 言語編	資料番号：U18513JJ1V0（第 1 版）
	CA850 Ver.3.20 C コンパイラ・パッケージ アセンブリ言語編	資料番号：U18514JJ1V0（第 1 版）
	CA850 Ver.3.20 C コンパイラ・パッケージ 操作編	資料番号：U18512JJ1V0（第 1 版）
	CA850 Ver.3.20 C コンパイラ・パッケージ リンク・ディレクティブ編	資料番号：U18515JJ1V0（第 1 版）
	PM+ Ver.6.30 プロジェクト・マネージャ ユーザーズ・マニュアル	資料番号：U18416JJ1V0（第 1 版）
	CA850 Ver.2.50 C コンパイラ・パッケージ コーディング・テクニック	資料番号：U16076JJ1V0（第 1 版）

1. 対象製品

CA850　（正式品名：CA703000）

2. 新たな制限事項

今回新たに，CA850 の制限事項 No.111，No.112 を追加しました。

- No.111　キャストを伴う比較演算が不正に最適化される制限
- No.112　代入文に対する不正移動の制限

3. 回避策

今回追加した制限事項の回避策は以下の通りです。詳細は別紙 1 を参照してください。

- No.111　不正最適化される比較演算の直前に「__asm("¥n");」を挿入してください。
- No.112　「__asm("¥n");」を挿入してください。（挿入が必要な行情報をチェックツールが出力します）

4. 修正した制限事項

CA850 V3.40 にて下記の制限事項を修正しました。詳細は別紙 1，別紙 2 を参照してください。

【 CA850 の制限事項 】

- No.104　ループの実行回数が不正になる制限
- No.105　文字列定数の内容が不正になる制限
- No.106　sscanf，fscanf，scanf 関数の型指定に関する制限
- No.107　atoi，atol，strtol，strtoul 関数の引数文字列に関する制限
- No.108　ビットフィールドをメンバに持つ構造体型変数の初期化の制限
- No.109　条件アセンブル擬似命令のネストの制限
- No.110　switch，if 文内での代入の制限

【 その他パッケージ・ツールの制限事項】

- No.9　PM+ C ソースのインクルードファイル情報取得の制限

5. 改善計画

今回追加した CA850 の制限事項 No.111，No.112 は，次バージョンで修正することを予定しています。

CA850 Ver.3.46 （2010 年 9 月末 リリース予定）

6. 制限事項一覧

制限事項の履歴とその詳細情報を制限事項一覧として別紙 1，別紙 2 に記載します。

7. 発行文書履歴

V850 コンパイラ・パッケージ CA850 使用制限事項 発行文書履歴

文書番号	発行日	記事
SBG-TT-0003	2001. 11. 2	統合版初版
SBG-TT-0064	2002. 2. 5	新規制限事項追加（No.68～No.71）
SBG-TT-0074	2002. 3. 7	新規制限事項追加（No.72）
SBG-TT-0154	2002. 7.12	新規制限事項追加（No.73～No.79）
SBG-TT-0218	2002. 10.10	制限事項修正 （No.2，No.3，No.14，No.28，No.43，No.44，No.60，No.71，No.73，No.74，No.75，No.76，No.77，No.78，No.79）
SBG-DT-03-0027	2003. 1.31	制限事項条件追加（No.56） 新規制限事項追加（No.80～No.82）
SBG-DT-03-0167	2003. 6.10	新規制限事項追加（No.83～No.87） その他パッケージ・ツール新規制限事項追加（No.1，No.2）
SBG-DT-03-0218	2003. 7.23	制限事項修正 （No.9，No.10，No.11，No.19，No.36，No.37，No.56，No.80，No.81，No.83，No.84，No.85，No.86，No.87，No.61 の一部）
SBG-DT-03-0254	2003. 9.26	誤記修正 • 「製品履歴」中の No.65，No.66 に誤記があったため修正 • No.9，No.10，No.11，No.19，No.36，No.37 の「改善策」に誤記があったため修正
SBG-DT-04-0005	2004. 1. 9	新規制限事項追加（No.88，No.89）
SBG-DT-04-0119	2004. 3.24	新規制限事項追加（No.90）
ZBG-CD-04-0009	2004. 5.28	制限事項修正 （No.1，No.35，No.59，No.61，No.88，No.89，No.90） 新規制限事項追加（No.91）
ZBG-CD-04-0070	2004. 9.22	新規制限事項追加（No.92）
ZBG-CD-05-0026	2005. 3.30	新規制限事項追加（No.93） その他パッケージ・ツール新規制限事項追加（No.3） 制限事項修正（No.47，No.48，No.91，No.92） 制限事項誤記修正（No.13）
ZBG-CD-05-0062	2005. 6.29	新規制限事項追加（No.94，No.95，No.96） 制限事項誤記修正（No.7）
ZBG-CD-05-0077	2005. 9. 1	その他パッケージ・ツール新規制限事項追加（No.4，No.5，No.6）
ZBG-CD-05-0112	2005. 12. 5	新規制限事項追加（No.97） その他パッケージ・ツール新規制限事項追加（No.7）
ZBG-CD-06-0044	2006. 5. 31	その他パッケージ・ツール新規制限事項追加（No.8） 制限事項修正（No.94，No.97） その他パッケージ・ツール制限事項修正（No.3，No.4，No.5，No.6，No.7）
ZBG-CD-07-0067	2007. 9.27	新規制限事項追加（No.98，No.99，No.100，No.101，No.102，No.103）
ZBG-CD-09-0025	2009. 5.21	CA850 新規制限事項追加（No.104，No.105，No.106，No.107，No.108，No.109，No.110） その他パッケージ・ツール新規制限事項追加（No.9） CA850 制限事項修正（No.93，No.98，No.99，No.100，No.101，No.102，No.103） その他パッケージ・ツール制限事項修正（No.8）
ZBG-CD-10-0030	2010. 9.13	CA850 新規制限事項追加（No.111，No.112） CA850 制限事項修正（No.104，No.105，No.106，No.107，No.108，No.109，No.110） その他パッケージ・ツール制限事項修正（No.9）

以 上

CA850 の制限事項一覧

1. 製品履歴

No.	仕様変更・追加／制限事項	バージョン									
		V2.41	V2.50	V2.6x	V2.70	V2.72	V3.00	V3.10	V3.20	V3.30	V3.4x
1	空白文字を含むファイル名の制限	×	×	×	○	○	○	○	○	○	○
2	アセンブリ・ソースの長さの制限	×	○	○	○	○	○	○	○	○	○
3	変数名の長さの制限	×	○	○	○	○	○	○	○	○	○
4	浮動小数点定数演算の演算時精度の制限	×	×	×	×	×	×	×	×	×	×
5	特定の定数演算処理の制限（Windows 版のみ）	○	○	○	○	○	○	○	○	○	○
6	関数の引数で構造体型の条件演算子の制限	×	×	×	×	×	×	×	×	×	×
7	関数の間接呼び出しの制限	×	×	×	×	×	×	×	×	×	×
8	意味不明な関数定義がエラーにならない制限	×	×	×	×	×	×	×	×	×	×
9	前処理命令と連続した数値がエラーにならない制限	×	×	○	○	○	○	○	○	○	○
10	前処理命令に続く不正な識別子がエラーにならない制限	×	×	○	○	○	○	○	○	○	○
11	union で定義した型を struct で指定してもエラーにならない制限	×	×	○	○	○	○	○	○	○	○
12	関数宣言で（ ）が余計に付いている場合エラーになる制限	×	×	×	×	×	×	×	×	×	×
13	extern をともなった構造体の型定義がエラーになる制限	○	○	○	○	○	○	○	○	○	○
14	文字列ストリングでの初期化の制限	×	○	○	○	○	○	○	○	○	○
15	無限ループの最後に switch-case 文の制限	○	○	○	○	○	○	○	○	○	○
16	デバッグ情報シンボルのエラーの制限	○	○	○	○	○	○	○	○	○	○
17	代入式の連続の制限	○	○	○	○	○	○	○	○	○	○
18	ランタイム・ライブラリのレジスタ割り付けの制限	○	○	○	○	○	○	○	○	○	○
19	セクション・ファイルとグローバル変数の制限	×	×	○	○	○	○	○	○	○	○
20	システム・コール ext_tsk の制限	○	○	○	○	○	○	○	○	○	○
21	セクション配置の制限	×	×	×	×	×	×	×	×	×	×
22	変数の実体がアセンブリ・ソースにあるセクション・ファイルの制限	×	×	×	×	×	×	×	×	×	×
23	同名の外部変数の仮定義が複数ファイルにあるセクション・ファイルの制限	×	×	×	×	×	×	×	×	×	×
24	ポインタ型定数を含む演算の制限	○	○	○	○	○	○	○	○	○	○
25	周辺 I/O レジスタのビットアクセスの制限	○	○	○	○	○	○	○	○	○	○
26	ネストした構造体，共用体の volatile 指定の制限	○	○	○	○	○	○	○	○	○	○
27	定数 0 の xor 演算の制限	○	○	○	○	○	○	○	○	○	○
28	V850E モードでのアセンブル最適化の制限	×	○	○	○	○	○	○	○	○	○
29	最適化オプション指定時のエラーの制限	×	×	×	×	×	×	×	×	×	×
30	最適化時のオブジェクト・サイズの制限	×	×	×	×	×	×	×	×	×	×

×：該当する
○：該当しない
－：対象外
※：チェックツールあり

No.	仕様変更・追加／制限事項	バージョン									
		V2.41	V2.50	V2.6x	V2.70	V2.72	V3.00	V3.10	V3.20	V3.30	V3.4x
31	最適化時のデバッグの制限	×	×	×	×	×	×	×	×	×	×
32	ループ内における_rcopy ルーチンの制限	○	○	○	○	○	○	○	○	○	○
33	構造体メンバのアドレスの制限	×	×	×	×	×	×	×	×	×	×
34	ビット・フィールドの制限	×	×	×	×	×	×	×	×	×	×
35	.option nomacro 疑似命令の制限	×	×	×	○	○	○	○	○	○	○
36	マクロ内に“~（チルダ）”のみの記述の制限	×	×	○	○	○	○	○	○	○	○
37	マクロがネストしている場合の‘~（チルダ）’の制限	×	×	○	○	○	○	○	○	○	○
38	不正命令 sal の制限	○	○	○	○	○	○	○	○	○	○
39	リンク・ディレクティブ・ファイル中のファイル名の制限	○	○	○	○	○	○	○	○	○	○
40	C ソース内における，固有シンボル，予約シンボルの参照の制限	×	×	×	×	×	×	×	×	×	×
41	リンク・オプション“-r”指定の制限（UNIX 版のみ）	○	○	○	○	○	○	○	○	○	○
42	EP 相対のセグメントに関するワーニングの制限	○	○	○	○	○	○	○	○	○	○
43	セグメントに関するワーニングの制限	×	○	○	○	○	○	○	○	○	○
44	アーカイブファイル中のオブジェクト・ファイル名の制限	×	○	○	○	○	○	○	○	○	○
45	romp850 の“-m オプション”による rompsec セクションのサイズの制限	○	○	○	○	○	○	○	○	○	○
46	パフォーマンス・チェッカの，出力ファイルのパス指定オプションの制限	×	－	－	－	－	－	－	－	－	－
47	クロス・レファレンス・ツールの，出力ファイルのパス指定オプションの制限	×	×	×	×	×	○	○	○	○	○
48	メモリ・レイアウト視覚化ツールの，出力ファイルのパス指定オプションの制限	×	×	×	×	×	○	○	○	○	○
49	&，|，~を使用した 1 ビット操作の制限	○	○	○	○	○	○	○	○	○	○
50	構造体パッキングでの unsigned short 型の制限	○	○	○	○	○	○	○	○	○	○
51	間接代入をはさんだ共用体の代入の制限	○	○	○	○	○	○	○	○	○	○
52	再リンク可能なオブジェクトの作成の制限	○	○	○	○	○	○	○	○	○	○
53	switch 文の-Wi,-O4 指定の制限	○	○	○	○	○	○	○	○	○	○
54	同名のグローバル関数とスタティック関数の制限	○	○	○	○	○	○	○	○	○	○
55	関数内の文字列定数の制限	○	○	○	○	○	○	○	○	○	○
56	変数の初期化の制限	×	×	○	○	○	○	○	○	○	○
57	引数に構造体を持つ関数ポインタ宣言の制限	○	○	○	○	○	○	○	○	○	○
58	sld／sst 命令のディスプレースメントの，ラベルの減算の制限	○	○	○	○	○	○	○	○	○	○
59	ROM 化プロセッサの内蔵 ROM チェックの制限	×	×	×	○	○	○	○	○	○	○
60	一時ファイル作成ディレクトリの制限	×	○	○	○	○	○	○	○	○	○

×：該当する
○：該当しない
－：対象外
※：チェックツールあり

No.	仕様変更・追加／制限事項	バージョン									
		V2.41	V2.50	V2.6x	V2.70	V2.72	V3.00	V3.10	V3.20	V3.30	V3.4x
61	コマンドファイル内の多バイト文字／半角カナ文字の制限	×	×	×	○	○	○	○	○	○	○
62	構造体パッキング時の構造体ポインタの制限	○	○	○	○	○	○	○	○	○	○
63	構造体への代入，参照の制限	○	○	○	○	○	○	○	○	○	○
64	関数間を跨いだ最適化処理の制限	○	○	○	○	○	○	○	○	○	○
65	unsigned char 型変数の tidata_word セクションへの配置の制限	○	○	○	○	○	○	○	○	○	○
66	前処理命令での定数式の制限	○	○	○	○	○	○	○	○	○	○
67	アセンブラ最適化処理でのコード移動の制限	○	○	○	○	○	○	○	○	○	○
68	最適化処理時の異常終了の制限	○	○	○	○	○	○	○	○	○	○
69	プロジェクト・マネージャのビルド時の制限	○	○	○	○	○	○	○	○	○	○
70	#pragma section 内の static 関数記述の制限	○	○	○	○	○	○	○	○	○	○
71	オフセット付きアドレスへのアクセスでのポインタ型定数の制限	×	○	○	○	○	○	○	○	○	○
72	if 文の最適化処理の制限	○	○	○	○	○	○	○	○	○	○
73	switch 文の最適化の制限	×	○	○	○	○	○	○	○	○	○
74	ビット・フィールドの初期化の制限	×	○	○	○	○	○	○	○	○	○
75	自動変数配列の初期化の制限	×	○	○	○	○	○	○	○	○	○
76	機種依存最適化部の ov フラグの制限	×	○	○	○	○	○	○	○	○	○
77	機種依存最適化部のシフトの制限	×	○	○	○	○	○	○	○	○	○
78	整数とポインタの共用体の制限	×	○	○	○	○	○	○	○	○	○
79	構造体パッキング指定時のビット・フィールド・メンバへの複合代入の制限	×	○	○	○	○	○	○	○	○	○
80	整数とポインタの共用体の最適化に関する制限	×	×	○	○	○	○	○	○	○	○
81	パッキングされた構造体引数の制限	×	×※	○	○	○	○	○	○	○	○
82	データ・スワップ組み込み関数の制限	×	○	○	○	○	○	○	○	○	○
83	構造体を引数とする関数のインライン展開の制限	×	×※	○	○	○	○	○	○	○	○
84	関数ポインタの制限	×	×※	○	○	○	○	○	○	○	○
85	組み込み関数__sasf の制限	－	×	○	○	○	○	○	○	○	○
86	float 型から short／unsigned short 型への変換の制限	×	×※	○	○	○	○	○	○	○	○
87	割り込み関数の制限	×	×※	○	○	○	○	○	○	○	○
88	複雑な計算式のコード生成における制限	×	×※	×※	○	○	○	○	○	○	○
89	インライン関数の引数に関する制限	×	×※	×※	○	○	○	○	○	○	○
90	ループ処理の最適化に関する制限	×	×※	×※	○	○	○	○	○	○	○

×：該当する
○：該当しない
－：対象外
※：チェックツールあり

No.	仕様変更・追加／制限事項	バージョン									
		V2.41	V2.50	V2.6x	V2.70	V2.72	V3.00	V3.10	V3.20	V3.30	V3.4x
91	if 文と goto 文によって未定義シンボルエラーが発生する制限	−	−	−	×	○	○	○	○	○	○
92	フラッシュ／外付け ROM 再リンク機能におけるリンク時の制限	−	−	−	×※	○	○	○	○	○	○
93	E2288 エラーの制限	−	×	×	×	×	×	×	○	○	○
94	セキュリティ ID とオプションバイトの制限	−	−	×	×	×	×	○	○	○	○
95	浮動小数点定数と汎整数型の制限	×	×	×	×	×	×	×	×	×	×
96	入出力関数の標準ライブラリの入力変換の制限	×	×	×	×	×	×	×	×	×	×
97	機種依存最適化部の代入の制限	×	×※	×※	×※	×※	×※	○	○	○	○
98	自動変数領域の一部が不正に削除される制限	×	×※	×※	×※	×※	×※	×※	○	○	○
99	hx850 の“-U オプション”による指定アドレスの制限	×	×	×	×	×	×	×	○	○	○
100	strcmp() と strncmp() の動作が異なる制限	×	×	×	×	×	×	×	○	○	○
101	65 文字以上の文字列定数の制限	×	×	×	×	×	×	×	○	○	○
102	アセンブラ最適化の制限	−	×	×	×	×	×	×	○	○	○
103	アドレスを取られた自動変数への代入文が不正に削除される制限	×	×※	×※	×※	×※	×※	×※	○	○	○
104	ループの実行回数が不正になる制限	×	×※	×※	×※	×※	×※	×※	×※	×※	○
105	文字列定数の内容が不正になる制限	×	×※	×※	×※	×※	×※	×※	×※	×※	○
106	sscanf，fscanf，scanf 関数の型指定に関する制限	×	×	×	×	×	×	×	×	×	○
107	atoi，atol，strtol，strtoul 関数の引数文字列に関する制限	×	×	×	×	×	×	×	×	×	○
108	ビットフィールドをメンバに持つ構造体型変数の初期化の制限	×	×※	×※	×※	×※	×※	×※	×※	×※	○
109	条件アセンブル擬似命令のネストの制限	×	×	×	×	×	×	×	×	×	○
110	switch，if 文内での代入の制限	×	×※	×※	×※	×※	×※	×※	×※	×※	○
111	キャストを伴う比較演算が不正に最適化される制限	○	○	○	×※	×※	×※	×※	×※	×※	×※
112	代入文に対する不正移動の制限	×	×※	×※	×※	×※	×※	×※	×※	×※	×※

×：該当する
○：該当しない
−：対象外
※：チェックツールあり

2. 使用制限事項の詳細

No. 1　空白文字を含むファイル名の制限

【内　容】　空白文字を含むファイル名，または，ディレクトリ名を使用することはできません。

【回避策】　空白文字を含むファイル名，または，ディレクトリ名を使用しないでください。

【改善策】　Ver.2.70 で修正しました。

No. 2　アセンブリ・ソースの長さの制限

【内　容】　大変長い変数名を扱った C ソースプログラムから出力されるアセンブリ・ソースで，1 行あたり 1024 文字を越えるとエラーとなります。

【回避策】　変数名等を短くして，1 行の長さが短くなるようにしてください。

【改善策】　Ver.2.50 で修正しました。

No. 3　変数名の長さの制限

【内　容】　1022 文字の変数名を，-g オプションを指定してコンパイルすると，アセンブラで 1023 文字になるため，アセンブラでエラーになります。

【回避策】　変数名を短くしてください。

【改善策】　Ver.2.50 で修正しました。

No. 4　浮動小数点定数演算の演算時精度の制限

【内　容】　整数へのキャストをともなうコンパイルにおいて，演算ができてしまうような浮動小数点演算を記述した場合，極微少の精度ずれが発生し，整数へのキャストにより不正な値になることがあります。なお，浮動小数点のまま扱う場合，問題は発生しません。

（例）`(long)(1.12 * 100);`

【回避策】　下記のいずれかのように変更してください。

```
float  f=1.12;
(long)(f * 100);
```

または，

```
float  f;
(long)(f=1.12 * 100);
```

【改善策】　制限事項とします。

No. 5　特定の定数演算処理の制限（Windows 版 のみ）

【内　容】 -2147483648/-1 あるいは-2147483648%-1 の式が最適化のフロー解析により発生する場合，コンパイル中に「このプログラムは不正な処理を行ったので強制終了されます」との Windows ダイアログが表示されます。

（例）
```
void f(){
    int  int_min = -2147483648;
    unsigned int ans1 = int_min/-1;
    unsigned int ans2 = int_min%-1;
}
```

【回避策】 下記のいずれかのように変更してください。

1. 計算結果を直接記述する

```
void f(){
     unsigned int ans1 = 2147483648;
     unsigned int ans2 = 0;
}
```

2. -2147483648 を const 変数以外の外部変数に入れて計算する

```
int int_min = -214783648;
void f(){
     unsigned int ans1 = int_min/-1;
     unsigned int ans2 = int_min%-1;
}
```

【改善策】 Ver.2.40 で修正しました。

No. 6　関数の引数で構造体型の条件演算子の制限

【内　容】 引数に構造体型の条件演算子が存在する場合に正しい分岐コードが生成されません。

（例）
```
typedef struct {int i;}S;
S ss1, ss2;
int  j;
void func() {
     func_call((j>10)?ss1:ss2);
}
```

【回避策】 下記のように if 文に変更してください。

```
if (j > 10){
     func_call(ss1);
} else {
     func_call(ss2);
}
```

【改善策】 制限事項とします。

No. 7　関数の間接呼び出しの制限

【内　容】 関数の間接呼び出し式が，オフセットを必要とする呼び出し方が，以下のいずれかのコンパイラ内部エラーとなります。

（メッセージ）

C2000：internal：gen_binary()：OP_CALL：left-child's operator is wrong

C5211: syntax error at line <num> in intermediate file

（例）
```
struct S{;
    int dummy;
    int func_body[0x100];
}sobj;
void f() {
    ((void(*)())sobj.func_body)();
}
```

【回避策】 下記のようにオフセット計算式と呼び出し式を分けてください。

```
void f() {
    void(*fp)()=(void(*)())&sobj.func_body;
    fp();
}
```

【改善策】 制限事項とします。

No. 8　意味不明な関数定義がエラーにならない制限

【内　容】 次のような意味不明な関数定義がエラーになりません。

（例）
```
typedef int INTFN();
INTFN  f{return(0);}
```

【回避策】 該当するようなコーディングはしないでください。

【改善策】 制限事項とします。

No. 9　前処理命令と連続した数値がエラーにならない制限

【内　容】 前処理命令と連続した数値の記述がエラーになりません。

（例） `#if0`

【回避策】 該当するようなコーディングはしないでください。

【改善策】 Ver.2.60 で修正しました。

No. 10　前処理命令に続く不正な識別子がエラーにならない制限

【内　容】 前処理命令に続く不正な識別子がエラーになりません。

（例） `#if $$$`

【回避策】 該当するようなコーディングはしないでください。

【改善策】 Ver.2.60 で修正しました。

No. 11　union で定義した型を struct で指定してもエラーにならない制限

【内　容】　union で定義した型を struct で指定してもエラーになりません。

（例）
```
union uu { char a; int b; };
struct uu uu;
```

【回避策】　該当するようなコーディングはしないでください。

【改善策】　Ver.2.60 で修正しました。

No. 12　関数宣言で（）が余計に付いている場合エラーになる制限

【内　容】　関数宣言で()が余計に付いている場合，エラーになります。

（例）
```
typedef int Int;
void f1((Int));
```

【回避策】　記述を以下のように変更してください。

```
typedef int Int;
void f1(Int);
```

【改善策】　制限事項とします。

No. 13　extern をともなった構造体の型定義がエラーになる制限

【内　容】　extern をともなった構造体の型定義がエラーになります。

（例）
```
extern struct tag { int i; };
```

【回避策】　extern を外して記述ください。

```
struct tag { int i; };
```

【改善策】　Ver.2.20 で修正しました。

No. 14　文字列ストリングでの初期化の制限

【内　容】　char 型および unsigned char 型以外へキャストした文字列ストリングに対し，その文字列リテラルのコードが出力されないため，内部エラーが出力され，終了してしまいます。

（メッセージ）
C7308：undefined block number(xx) reference in opcode(xx)

（例）
```
struct S{;
    long l;
}sobj = {(long)"string"};
```

【回避策】　次のように変更してください。

```
char c[]="string";
struct S{;
    long l;
}sobj = {(long)c};
```

【改善策】　Ver.2.50 で修正しました。

No. 15　無限ループの最後に switch-case 文の制限

【内　容】 -Ot，-g 指定時に，無限ループが switch-case 文で終了する場合（switch-case 文の後ろが不要処理として削除される場合も含みます）に，内部エラーが出力されます。switch-case 文が，switch-table 分岐となる場合にのみ発生します。

（メッセージ）

C6101：illegal intermediate file(operator)

【回避策】 以下のいずれかの方法で回避してください。

1. -Xcase=ifelse オプションを指定し，switch-table 分岐のコードを出力しないようにする
2. -Xcase=binary オプションを指定し，switch-table 分岐のコードを出力しないようにする
3. 無限ループの最後に__asm("￥n")を記述する

【改善策】 Ver.2.41 で修正しました。

No. 16　デバッグ情報シンボルのエラーの制限

【内　容】 最適化指定なしで，かつ，デバッグ情報出力（-g）を指定していた場合に，外部あるいは静的変数がレジスタに割り付いて，割り付いた領域が以下のいずれかで終了する場合にリンカで undefined symbol になります。

- 2^n+1 との乗算
- 2^n-1 との乗算

【回避策】 以下のいずれかの方法で回避してください。

1. 最適化（-Os または-Ot）を指定する
2. 対象となる外部あるいは静的変数を volatile 宣言する

【改善策】 Ver.2.40 で修正しました。

No. 17　代入式の連続の制限

【内　容】　次の条件を満たしている場合に，必要なコードが削除されてしまいます。

1. -Os または-Ot オプション指定時の広域最適化処理において，テンポラリレジスタの不足が発生する場合
2. 上記 1 が発生している場合，広域最適化処理後のコード生成フェーズで，テンポラリレジスタを一度スタックに退避し，必要に応じてスタックからテンポラリレジスタに復帰させる処理が行われるが，この状態で下記に示す C 言語記述を検出した場合には，上記の復帰処理コードが生成されない。
3. 上記 1，2 項が重なった場合のみ，コード生成フェーズの後の機種依存最適化フェーズにて，不要な命令列として認識され，本来必要なコードが削除されるため，今回の該当不具合となる。

（記述）

```
<val> <op2>= (<val_1>) <op1> <imm_1>;
<val> <op2>= (<val_2>) <op1> <imm_2>;
                :
<val> <op2>= (<val_n-1>) <op1> <imm_n-1>;
<val> <op2>= (<val_n>) <op1> <imm_n>;
```

（説明）

<op1>,<op2>：演算子
<val>：変数
<imm>：直値
<val_n>：変数，あるいは，変数と変数の演算式
<n>：5〜10 以上（関数の規模，変数の使用状況によります）
op1：+，&，|，^，-
『-』は，符号反転して add 命令を使うケースに限ります。
op1 のオペランドは左右逆でも発生します。
op2：+，*，&，|，^
val：int，unsigned，long，unsigned long の非 volatile の同一変数

（例）

```
x = a;
x |= b & 0x000000f0;
x |= c & 0x00000f00;
x |= d & 0x0000f000;
x |= e & 0x000f0000;
x |= f & 0x00f00000;  /* コードが出力されない */
x |= g & 0x0f000000;  /* コードが出力されない */
x |= h & 0xf0000000;  /* コードが出力されない */
```

【回避策】　以下のいずれかの方法で回避してください。

1. 該当する C 言語記述があるソース・ファイルについてのみ，最適化オプション（-Os または-Ot）を指定しないで下さい。
2. 次のように C 言語記述を変更して，不具合に該当しない様な記述にしてください。

```
x  =  a              |
      b & 0x000000f0 |
      c & 0x00000f00 |
      d & 0x0000f000 |
      e & 0x000f0000 |
      f & 0x00f00000 |
      g & 0x0f000000 |
      h & 0xf0000000;
```

【改善策】　Ver.2.40 で修正しました。

No. 18　ランタイム・ライブラリのレジスタ割り付けの制限

【内　容】 乗除算がランタイム・ライブラリで出力された結果，レジスタ割り付けが不正になり，r6 レジスタ，または，r7 レジスタの値が壊れることがある。条件は次の 5 つの条件をすべて満たしている場合です。

1. V850 コアのデバイスを指定
2. -Os あるいは-Ot を指定
3. 乗算の乗数または除算の除数が次のいずれかの場合
   - “ signed char ” または “ unsigned char ” の変数と，“ -16384 以上 ”“ 16383 以下 ” の定数との加算または減算
   - 以下の一方が変数で，他方が指定した型の範囲内に入る定数の乗算
     - signed char と signed char
     - signed char と unsigned char
     - unsigned char と signed char
4. 上記 3 の定数がレジスタに割り付く
5. 上記 3 の乗算あるいは除算の実行時生存している変数，あるいは定数が r6 あるいは r7 に割りつく

（例）

```
char s,t;
void func(int a,int b){
int c;
    c = ( s - 0xaa) * t;
    if ( b < 0xaa) c = 0;
    func2(c);
}
```

【回避策】 以下のいずれかのオプションで回避してください。

1. -Wo,-preg0（引数用レジスタにレジスタ割り付けを行なわない指定をします）
2. -Wo,-XTm （mulh／divh を出力しない指定をします。）

【改善策】 Ver.2.40 で修正しました。

No. 19　セクション・ファイルとグローバル変数の制限

【内　容】 グローバル変数で，セクション・ファイルとソース・ファイルで，セクションの不整合が起きた際に警告メッセージが出ない場合があります。

【回避策】 セクション・ファイルを変更する場合は，セクションの不整合が発生しないよう注意して下さい。

【改善策】 Ver.2.60 で修正しました。

No. 20　システム・コール ext_tsk の制限

【内　容】　“ #pragma rtos_task ”を指定してリアルタイム OS のタスクを記述し，そのタスク内でシステム・コール“ ext_tsk ”を使用していた場合，-g オプションを指定してコンパイルするとエラーが発生し，異常終了する場合があります。

（例）

```
extern  int  func1(void);
#pragma rtos_task usrtsk
void usrtsk(int Param)
{
    int  i;
    i = func1();
    ext_tsk();
}
```

【回避策】　下記の例のように，ext_tsk システム・コールを，関数ポインタを使って間接的に呼び出すように変更して下さい。

```
extern  int  func1(void);
#pragma rtos_task usrtsk
void usrtsk(int Param)
{
    int  i;
    void (*ext_tsk_ptr)() = ext_tsk;
    i = func1();
    (*ext_tsk_ptr)();
}
```

【改善策】　Ver.2.30 で修正しました。

No. 21　セクション配置の制限

【内　容】　#pragma section 指令による tidata セクション配置指定，または sf850 を用いて tidata セクションに配置指定した「構造体の char 型配列」や「char 型メンバへのアクセス」において，その配列，またはメンバのアクセスに使用する sst 命令／sld 命令のディスプレースメント値を越えた場合，リンカでエラーを発生します。

【回避策】　以下のいずれかの方法で回避してください。

1. リンカのエラーとなる構造体の char 型メンバや char 型配列を使用しないでください。
2. リンカのエラーとなる構造体の char 型メンバや char 型配列を tidata セクションに割り当てないでください。

【改善策】　制限事項とします。

No. 22　変数の実体がアセンブリ・ソースにあるセクション・ファイルの制限

【内　容】　変数の実体がアセンブリ・ソースにあり，その変数を C ソース上から参照しているアプリケーションにおいて，セクション・ファイルを sf850 で生成していた場合，リンク時にエラーが発生します。

【回避策】　アセンブリ・ソース中にある変数を，セクション・ファイルから削除して下さい。

【改善策】　制限事項とします。

No. 23　同名の外部変数の仮定義が複数ファイルにあるセクション・ファイルの制限

【内　容】 同名の外部変数の仮定義が複数ファイルにあるとき，セクション・ファイルを sf850 で生成していた場合，リンク時にシンボルの二重定義になる場合がある。

（メッセージ）

ld850：fatal error: symbol "_xxxx" multiply defined.

【回避策】 同名の外部変数の仮定義が複数ある場合は，外部変数を参照するファイルでは，必ず extern 宣言してください。

【改善策】 制限事項とします。

No. 24　ポインタ型定数を含む演算の制限

【内　容】 ポインタ型定数と複数の任意の式を含む演算において，不正なコードが出力されます。

（例）
```
int   i1;
int   i2;
char  c;
c = (char *)10 + i1 + i2 + 10;
```

【回避策】 演算を括弧で囲み，そのあとにポインタにキャストしてください。

```
c = (char *)( 10 + il + i2 + 10 );
```

【改善策】 Ver.2.40 で修正しました。

No. 25　周辺 I/O レジスタのビットアクセスの制限

【内　容】 周辺 I/O レジスタにビットアクセスで代入される変数もしくは式が，レジスタ割り付けされた場合に，コード生成部（cgen）が不正コードを出力することがあります。

（例）
```
void func(unsigned char a)
{
    int i = 8;
    while(i){
        P0.0 = a;
        a >>= 1;
        i--;
    };
}
```

【回避策】 周辺 I/O レジスタにビットアクセスで代入する場合には，変数ではなく値を代入してください。

```
void func(unsigned char a)
{
    int i = 8;
    while(i){
        if(a & 0x1){
            P0.0 = 1;
        } else {
            P0.0 = 0;
        }
        a >>= 1;
        i--;
    };
}
```

【改善策】 Ver.2.40 で修正しました。

No. 26　ネストした構造体，共用体の volatile 指定の制限

【内　容】　構造体（共用体）中に入れ子になった構造体（共用体）メンバへのアクセスが，volatile 宣言されたポインタ変数で，かつ，複合代入演算の左オペランドに書かれた場合に，volatile が無効になります。

（例）
```
typedef union {
    struct {
        unsigned char a0;
        unsigned char a1;
        unsigned short a2;
    }a;
    unsigned long c;
}T;
#define s(*(volatile T *)0x0100000)
s.a.a1 |= 0x1;{
```

【回避策】　複合代入を単純代入演算に変更してください。

```
s.a.a1 = s.a.a1 | 0x1;
```

【改善策】　Ver.2.40 で修正しました。

No. 27　定数 0 の xor 演算の制限

【内　容】　最適化（-Os または -Ot）指定時に，定数 0 との xor 演算で not 命令が出力されます。

（例）
```
a = 0;
a ^= b;
```

【回避策】　定数 0 との xor 演算の場合には，変数をそのまま代入するように変更してください。

```
a = b;
```

【改善策】　Ver.2.40 で修正しました。

No. 28　V850E モードでのアセンブル最適化の制限

【内　容】　V850E へ対応していますが，V850E モードではアセンブラの最適化は動作しません。

【回避策】　ありません。

【改善策】　Ver.2.50 で修正しました。

No. 29　最適化オプション指定時のエラーの制限

【内　容】　コンパイル時に指定する最適化オプションの最適化レベルを高くすることにより，コンパイル中に実行されるフェーズ（最適化機能やコンパイル機能など）が多くなります。-Ot オプションを指定した場合，このフェーズ間で生成される中間ファイルが巨大になり，fatal error を起こす場合があります。

【回避策】　最適化オプションのレベルを下げて（-Os など）コンパイルして下さい。

【改善策】　制限事項とします。

No. 30　最適化時のオブジェクト・サイズの制限

【内　容】　最適化を指定し，デバッグ情報を含むオブジェクト・ファイルのサイズは，非常に大きくなる場合があります。

【回避策】　最適化レベルを下げるか，デバッグしたいファイルだけにデバッグ情報を出力する“ - g オプション”をつけてコンパイルするようにしてください。

【改善策】　制限事項とします。

No. 31　最適化時のデバッグの制限

【内　容】　最適化を指定して，ソースデバッグを行なう際は次の制限事項があります。

1. 変数の値を参照した際，正しい値でなく，計算途中の一時的な値が得られることがあります。
2. 配列の一部，構造体の要素，ユーザー定義型のポインタ変数がレジスタに割りつけられた場合，デバッガの変数ウィンドウ等にて変数の表示／変更が不正になることがあります。
3. 自動変数の配列の一部，構造体の要素が使用されない場合，領域が削除されることがあります。この場合，デバッガの変数ウィンドウ等にて変数の表示／変更が不正になることがあります。変更の場合，スタックを破壊する可能性があります。

【回避策】　ありません。

【改善策】　制限事項とします。

No. 32　ループ内における _rcopy ルーチンの制限

【内　容】　_rcopy ルーチンを while 文などのループ内で使用した場合，-Os 以上の最適化を行なうとコンパイラは内部エラーを発生します。

（例 1）
```
while (_rcopy(&_S_romp, -1) < 0) {
                ;
 }
```
（例 2）
```
for ( ; ; ) {
        ret = _rcopy (&_S_romp, -1);
        if (ret == 0) {
                break;
        }
 }
```

【回避策】　_rcopy ルーチンを呼び出す関数を作成し，その関数をループで使用するようにしてください。

```
void main(void)
{
        while (dummy() < 0) {
                ;
        }
}
int dummy() {
        return _rcopy(&_S_romp, -1) ;
}
```

【改善策】　Ver.2.30 で修正しました。

No. 33　構造体メンバのアドレスの制限

【内　容】 構造パッキングを行って，以下のいずれかの条件に該当する場合，

1. ミス・アライン・アクセスに対応していないデバイスを使用している
2. ミス·アライン・アクセスに対応したデバイスで，ミス・アライン・アクセスを禁止している

データのアクセスは，デバイスのデータ・アライメントに従い，アドレスをマスクしてアクセスされるため，構造体メンバのアドレスでのアクセスで，データの消失や切り捨てが生じます。

（例）
```
struct test {
    char c;     /* offset 0 */
    int  i;     /* offset 1-4 */
} test;
int *ip, i;
void func(){
    i = *ip;    /* マスクされたアドレスでアクセスされる */
}
void func2(){
    ip = &(test.i);
}
```

【回避策】 ありません。

【改善策】 制限事項とします。

No. 34　ビット・フィールドの制限

【内　容】 構造パッキングにおけるビット・フィールドへのアクセスにおいて，ビット・フィールドへのアクセスではビット・フィールドの幅がメンバの型以下の場合，メンバの型で読み込むため，オブジェクトの外部（データのない領域）もアクセスします。通常，正常に実行されますが，I/O がマッピングされていると，不正アクセスとなる場合があります。

（例）
```
struct S {
    int  x:21;
} sobj;         /* 3 バイト */
sobj.x = 1;
```

【回避策】 ありません。

【改善策】 制限事項とします。

No. 35　.option nomacro 疑似命令の制限

【内　容】 .option nomacro 疑似命令の指定範囲中にある mov 命令が syntax error になる場合があります

（例）
```
.text
.option nomacro
mov 0x12345678, r10
```

【回避策】 該当する mov 命令を.option nomacro 疑似命令の指定範囲からはずしてください。

【改善策】 Ver.2.70 で修正しました。

No. 36　マクロ内に '~（チルダ）' のみの記述の制限

【内　容】 マクロ内に '~' のみを記述した場合アセンブラが異常終了します。

（例）
```
.macro MACRO
~
.endm
MACRO
```

【回避策】 このような記述をしないでください。

【改善策】 Ver.2.60 で修正しました。

No. 37　マクロがネストしている場合の "~（チルダ）" の制限

【内　容】 マクロがネストしている場合に "~（チルダ）" がエラーになる場合があります。

（例）
```
    .macro  inc.w  var, reg
            ld.w   var, reg
            add    1, reg
            st.w   reg, var
    .endm
    .macro aaa    rnum
            ld.w   [r~rnum], r30
            add    1, r30
            st.w   r30, [r~rnum]
    .endm
    .macro bbb    rnum
            inc.w  [r~rnum], r30
    .endm
    .set   pathreg, 25
    .text
    .globl  f
f:
    aaa   $pathreg    -- OK
    bbb   $pathreg    -- Error

    .macro MACRO
```

【回避策】 マクロをネストさせないでください。

【改善策】 Ver.2.60 で修正しました。

No. 38　不正命令 sal の制限

【内　容】 デバイスに存在しない命令"sal"を記述しても，アセンブラでエラーとならず，不正なコードを出力します。ただし，コンパイラではこの命令は出力しません。

（例）
```
sal   3, r10
```

【回避策】 ありません。ソースを記述する際に注意してください。

【改善策】 Ver.2.40 で修正しました。

No. 39　リンク・ディレクティブ・ファイル中のファイル名の制限

【内　容】　マッピング・ディレクティブにおけるファイル名指定で，{ } 内に指定するファイル名が A，F，H，L，V（小文字も含む）で始まり，2 文字目が数字である文字列の場合，リンカでエラーになります。

（メッセージ）

ld850：syntax error :line num：string is illegal in file specification field

【回避策】　ファイル名を上記の条件に当てはまらないようにしてください。

【改善策】　Ver.2.30 で修正しました。

No. 40　C ソース内における，固有シンボル，予約シンボルの参照の制限

【内　容】　C ソース内において，__gp_DATA などのターゲット固有シンボル，__stext などの予約シンボルを参照することができません。

【回避策】　C ソース内では，ターゲット固有シンボルや予約シンボルを使用しないで下さい。

【改善策】　制限事項とします。

No. 41　リンク・オプション "-r" 指定の制限（UNIX 版のみ）

【内　容】　UNIX 版において，リンク時に "-r オプション" を指定したとき，以下のパターンが重なった場合 core dump することがあります。

1. TEXT 属性セクションが存在しない
2. .sdata／.sbss セクションにデータを配置している
3. 配置属性ありセクション（ディレクティブで“ ?A ”指定あり）かつ，初期値なし（$NOBITS）セクションのサイズが奇数のものがある。
4. ld850 -r で一旦リロケータブルオブジェクトを作成し，再度リンクを行なう。

【回避策】　以下のいずれかの方法で回避してください。

1. ld850 の -r オプションを指定しない
2. 初期値なしセクションサイズを偶数とする

（例）

（修正前）

```
#pragma section sidata begin
static unsigned char a;
#pragma section sidata end
static unsigned long b;
```

（修正後）

```
#pragma section sidata begin
static unsigned char a;
static unsigned char dummy;    ← ダミーを追加
#pragma section sidata end
static unsigned long b;
```

3. char 型変数に初期値を与える

（例）

（修正前）

```
#pragma section sidata begin
static unsigned char a;
#pragma section sidata end
static unsigned long b;
```

（修正後）

```
#pragma section sidata begin
static unsigned char a = 0;    ← 初期値を与える
#pragma section sidata end
static unsigned long b;
```

【改善策】　Ver.2.30 で修正しました。

No. 42　EP 相対のセグメントに関するワーニングの制限

【内　容】　次の条件が全て該当する場合に，ld850 で次のワーニングが出力されます。

1. V850E のデバイスを品種指定
2. そのデバイスの内蔵 RAM の先頭アドレスが 0x03xxxxxx ではない
3. SEDATA セグメントまたは SIDATA セグメントが存在する

（メッセージ）
ld850：warning: segment "SIDATA" (0x03xxxxxx-0x03xxxxxx) must be in
EP-relative-address-able range (0x0xxxxxxxc000-0x0xxxxxxx).

【回避策】　リンク・マップを確認して内蔵 RAM のアドレスの前後に SEDATA セグメントと SIDATA セグメントが配置されているのを確認して，上記ワーニングを無視してください。

【改善策】　Ver.2.40 で修正しました。

No. 43　セグメントに関するワーニングの制限

【内　容】　V850E/MA1 や V850E/IA1 のように，28 ビット空間を持つデバイスを品種指定し，セクション配置をし，そのアドレスを 26 ビットマスクした値が配置禁止アドレスに重なる場合に，次の不要なワーニングを出力します。メッセージは出力されますが，生成するオブジェクトは問題ありません。

（メッセージ）
ld850: warning: segment "XXXX" overflowed highest address of target machine.

（例）内蔵 ROM が 0x00000000～0x0003ffff，外部メモリが 0x00100000 からの場合には，0x00040000～0x000fffff は，配置禁止になります。しかし，

AREA1：0x?4040000～0x?40fffff
AREA2：0x?8040000～0x?80fffff
AREA3：0x?c040000～0x?c0fffff

に関しては，配置可能になるため，不要なメッセージとなります。具体的には，内蔵 ROM 直後と外部メモリとの間の空間の AREA1，2，3 に相当するアドレスが不要に出力されます。

【回避策】　リンク・マップを確認して内蔵 RAM のアドレスの前後に SEDATA セグメントと SIDATA セグメントが配置されているのを確認して，上記ワーニングを無視してください。

【改善策】　Ver.2.50 で修正しました。

No. 44　アーカイブファイル中のオブジェクト・ファイル名の制限

【内　容】　アーカイブファイル中のオブジェクト・ファイル名が 15 文字以上の場合に，以下の現象が発生します。

1. リンク・ディレクティブ・ファイルの指定どおりにマッピングできない。
2. リンク・マップ，リンクエラーメッセージ中のファイル名表示が異常になる。

【回避策】　アーカイブファイルを作成時には，オブジェクト・ファイル名を 15 文字未満にしてください。

【改善策】　Ver.2.50 で修正しました。

No. 45　romp850 の “ -m オプション ” による rompsec セクションのサイズの制限

【内　容】 romp850 の “ -m オプション ” を使用して表示されるメモリマップで，rompsec セクションのトータル・サイズ表示に誤りがあります。コピー情報セクションのサイズが余分に足し込まれており，実際のサイズよりも大きくなっています。

【回避策】 rompsec セクションサイズから，コピー情報セクションのサイズ分を引くことによって正しいサイズを取得して下さい。または dump850 を使用してダンプ情報を出力し，正しいサイズを確認してください。

【改善策】 Ver.2.30 で修正しました。

No. 46　パフォーマンス・チェッカの，出力ファイルのパス指定オプションの制限

【内　容】 出力するファイルを指定して，出力ファイルのパス指定オプション（-o）で解析結果出力先のフォルダを指定しても，指定したフォルダに出力せずに，カレントフォルダ（実行したフォルダ）に解析結果を出力します。-all オプションを指定している場合には，-o オプションは指定したフォルダに解析結果を出力します。

【回避策】 出力したいフォルダで実行してください。

【改善策】 Ver.2.50 以降では対象外です

No. 47　クロス・レファレンス・ツールの，出力ファイルのパス指定オプションの制限

【内　容】 出力するファイルを指定して，出力ファイルのパス指定オプション（-o）で解析結果出力先のフォルダを指定しても，指定したフォルダに出力せずに，カレントフォルダ（実行したフォルダ）に解析結果を出力します。-all オプションを指定している場合には，-o オプションは指定したフォルダに解析結果を出力します。

【回避策】 出力したいフォルダで実行してください。

【改善策】 Ver.3.00 で修正しました。

No. 48　メモリ・レイアウト視覚化ツールの，出力ファイルのパス指定オプションの制限

【内　容】 出力するファイルを指定して，出力ファイルのパス指定オプション（-o）で解析結果出力先のフォルダを指定しても，指定したフォルダに出力せずに，カレントフォルダ（実行したフォルダ）に解析結果を出力します。-all オプションを指定している場合には，-o オプションは指定したフォルダに解析結果を出力します。

【回避策】 出力したいフォルダで実行してください。

【改善策】 Ver.3.00 で修正しました。

No. 49　&，|，~を使用した 1 ビット操作の制限

【内　容】 最適化（-Os または -Ot）指定時に，次の 2 つの条件に当てはまるレジスタに割り付かなかった短整数型の変数に対して，&，|，~を使用した 1 ビット操作を行なうとコードが不正になります。

1. 非 volatile の自動変数（静的変数と引数は対象外）
2. 対象の変数より前で宣言された signed int か signed long の変数がレジスタに割り付いた場合

（例）
```
void f(void){
    int i;
    short s = 0;
    i = g(&s);
    switch(i){
        case 100:
            s &= 0xfffe;
            break;
        case 200:
            s |= 1;
            break;
        case 300:
            s ^= 1;
            break;
    }
    i = i / s;
}
```

【回避策】 以下のオプションで回避してください。
　-Wo,-XTb（ビット操作命令の適用範囲を抑止します）

【改善策】 Ver.2.40 で修正しました。

No. 50　構造体パッキングでの unsigned short 型の制限

【内　容】 V850E の品種指定し，構造体パッキング機能を使用し，unsigned short 型メンバが奇数番地に配置された場合に，不正なコードが出力されます。short 型の場合には，本不具合には該当いたしません。

（例）
```
#pragma pack(1)
struct s {
    unsigned char a;
    unsigned short b;
}*x;
x->b = 512;
```

【回避策】 unsigned short 型のメンバを signed short 型に変更してください。
型により変数の取りうる値に違いがありますので，注意してください。

【改善策】 Ver.2.41 で修正しました。

No. 51　間接代入をはさんだ共用体の代入の制限

【内　容】 “基本ブロック内の冗長な代入文の削除の最適化”において，次のような命令の並びの場合に，1 の代入が不正に削除されてしまいます。

1. アドレス演算子&を使用していない共用体への代入
2. 不定アドレスへの間接代入
   不定アドレスとして該当する記述は以下となります。
   - ポインタ
   - ポインタ以外の変数をキャストしてポインタにしたもの
   - 絶対アドレス
     ※ 不定アドレス：ポインタも変数なので，宣言しただけでは値を持ちません。そのためポインタの指すアドレスは不定な状態となります。このような不定なアドレスを指す状態のもの。
3. 共用体の 1 とオーバーラップするほかのメンバへの参照または代入
4. 1 の共用体のメンバへの代入

（例）
```
typedef union{
    unsigned char a[2];
    unsigned short b;
} U;
U t;
unsigned short *p;
t.a[1] = 1;      /* この代入コードが不正に削除される */
*p = 0;
t.b >>= 4;
t.a[1] = 3;
```

【回避策】 以下のオプションで回避してください。
　-Wo,-Nce（基本ブロック内の複写の伝播と共通部分式の削除の最適化を抑止します）

【改善策】 Ver.2.41 で修正しました。

No. 52　再リンク可能なオブジェクトの作成の制限

【内　容】 リンク時に，複数の入力ファイル中の各々にリロケーション解決を要する同名のセクションが存在し，-r 指定で再リンク可能なオブジェクト・ファイルを作成する段階で，リンク・ディレクティブ・ファイルを使用して異なるセクションに配置すると，不正なリロケーション情報を保持するオブジェクト・ファイルを生成します。このオブジェクトをリンクすると，リロケーション情報が不正であるためオブジェクト内の命令が不正に書き換えられます。

（例）
```
USRTEXT : !LOAD ?RX {
    .USRTEXT1 = $PROGBITS ?AX .USRTEXT{a.o(libusr.a)};
    .USRTEXT2 = $PROGBITS ?AX .USRTEXT{b.o(libusr.a)};
    .USRTEXT3 = $PROGBITS ?AX .USRTEXT{c.o(libusr.a)};
    .USRTEXT4 = $PROGBITS ?AX .USRTEXT{d.o(libusr.a)};
};
```

【回避策】 リンク・ディレクティブ・ファイルに，リロケーション情報セクションを入力ファイルごとに別々に配置されるように指定してください。リロケーション情報のセクション名は“.rela+セクション名”になります。セクションの属性は“$RELA ?N”になります。

（例）
```
USRTEXT    : !LOAD ?RX {
    .USRTEXT1 = $PROGBITS ?AX .USRTEXT{a.o(libusr.a)};
    .USRTEXT2 = $PROGBITS ?AX .USRTEXT{b.o(libusr.a)};
    .USRTEXT3 = $PROGBITS ?AX .USRTEXT{c.o(libusr.a)};
    .USRTEXT4 = $PROGBITS ?AX .USRTEXT{d.o(libusr.a)};
};

    .rela.USRTEXT1 = $RELA ?N .rela.USRTEXT{a.o(libusr.a)};
    .rela.USRTEXT2 = $RELA ?N .rela.USRTEXT{b.o(libusr.a)};
    .rela.USRTEXT3 = $RELA ?N .rela.USRTEXT{c.o(libusr.a)};
    .rela.USRTEXT4 = $RELA ?N .rela.USRTEXT{d.o(libusr.a)};
```

【改善策】 Ver.2.41 で修正しました。

No. 53　switch 文の-Wi,-O4 指定の制限

【内　容】 V850E のデバイスを品種指定し，switch 文がテーブルジャンプのコードになる場合[注]，-Wi,-O4 指定をすると，不正な最適化を行い，不正にコードが削除されてしまいます。

注：CA850 では，switch 文に対して次の 2 つの条件が同時に満たされる場合にテーブルジャンプのコードを出力します。

- case ラベルの個数が 4 個以上
- ラベルの値の下限と上限の差が case の数の 3 倍まで

【回避策】 -Wi,-O4 オプションを指定せずに，-Ol オプションを指定してください。

【改善策】 Ver.2.41 で修正しました。

No. 54　同名のグローバル関数とスタティック関数の制限

【内　容】 同名のグローバル関数とスタティック関数ある場合に，二重定義エラーになりません。そのため，不正なエラー・メッセージが出力されます。

1. 内部エラー（C7318：illegal label (xxxx)）が出力される
2. リンク時に不正なデバッグ情報シンボル（.Gxxxx）が undefined symbol エラーになる

（例）
```
static void func(){}
void func(){}
```

【回避策】 同名のグローバル関数とスタティック関数は定義しないようにしてください。

【改善策】 Ver.2.41 で修正しました。

No. 55　関数内の文字列定数の制限

【内　容】 文字列定数が関数内で定義された場合に，8200 文字を超えると内部エラーになります。

（メッセージ）

C2000：internal: string literal too long

（例）
```
void func(void)
{
    char *p = "abcde…";    /* 8200 文字を超えている場合 */
}
```

【回避策】 文字列定数を 8200 文字以下にしてください。

【改善策】 Ver.2.41 で修正しました。

No. 56　変数の初期化の制限

【内　容】 次のいずれかの条件を満たす場合に，エラー・メッセージを出力し，コンパイルを中止する場合があります。

1. 変数の初期化が多数ある
2. “デバッグ情報出力（-g）指定”と“変数の初期化”がある，かつ，次の情報が多い
   - ファイル名(インクルードファイルを含む)
   - 関数名
   - 変数名
   - 列挙のタグ名と列挙子名
   - 構造体，共用体のタグ名とメンバ名
   - typedef 名
   - ラベル名

（メッセージ）

C7317: illegal block no (xxxxxxxx)

【回避策】 ファイルを分割して，1 コンパイル当たりの初期化の数を少なくする，または，上記の情報を少なくするようにしてください。
2. の場合には，初期値をもつ配列等の定義部分のみを別ファイルにするだけで回避できる場合があります。

【改善策】 Ver.2.60 で修正しました。

No. 57　引数に構造体を持つ関数ポインタ宣言の制限

【内　容】 デバッグ指定時（-g オプション指定時）に，引数に構造体を持つ関数ポインタ宣言で，その構造体定義がない場合，デバッグ情報の.Gxxxx シンボルが，リンカ（ld850）で次のエラーとなります。

（メッセージ）

ld850：fatal error: undefined symbol.

（例）
```
void  *(*x)(struct SSS *);
```

【回避策】 関数ポインタ宣言よりも前に使用している構造体を定義してください。

（例）
```
struct SSS {
    unsigned char  s;
    unsigned char  t;
}sss_obj;
void  *(*x)(struct SSS *);
```

【改善策】 Ver.2.41 で修正しました。

No. 58　sld／sst 命令のディスプレースメントの，ラベルの減算の制限

【内　容】 sld／sst 命令のディスプレースメントにラベル同士の減算式を記述し，かつ，その式の値がアセンブラの命令展開により変動する可能性のある場合に，不正な命令になることがあります。なお，CA850 では該当するコードは生成しません。

（例）
```
LAB1:
    sld.w  LAB2 - LAB1, r10
              :
    mov    LAB2 - LAB1, r11  --  命令展開する可能性あり
LAB2:
```

【回避策】 ラベル同士の減算式ではなく，値を直接記述してください。

【改善策】 Ver.2.41 で修正しました。

No. 59　ROM 化プロセッサの内蔵 ROM チェックの制限

【内　容】 ROM 化プロセッサ（romp850）において，.text セクションの有無や配置アドレスによって下記のワーニング・メッセージが，出力されなくてもよい場合に出力されたり，出力しなければならない場合に出力されたりします。ただし，出力されるオブジェクトに問題はありません。

（メッセージ）

Warning: rompsec section overflowed highest address of target machine.

【回避策】 ROM 化プロセッサ（romp850）の“-m オプション”でマップを出力して，rompsec セクションが内蔵 ROM を越えていないかを確認してください。越えていない場合には，上記のワーニング・メッセージは無視してください。

【改善策】 Ver.2.70 で修正しました。

No. 60　一時ファイル作成ディレクトリの制限

【内　容】 プロジェクト・マネージャで一時ファイル作成ディレクトリとして相対パスを指定すると，プロジェクト・マネージャ用のコマンドファイル（pm*.cmd）が，一時ファイル作成ディレクトリに削除されずに残る場合があります。

【回避策】 プロジェクト・マネージャを終了後に，コマンドファイル（pm*.cmd）を削除してください。また，一時ファイル作成ディレクトリには，相対パスを指定しないでください。

【改善策】 Ver.2.50 で修正しました。

No. 61　コマンドファイル内の多バイト文字／半角カナ文字の制限

【内　容】 次のコマンドの，コマンドファイル内で多バイト文字／半角カナ文字を使用した場合，文字列の区切り文字の識別に失敗してエラーになります

1. as850　（Ver.2.60 で修正しました）
2. ar850　（Ver.2.50 で修正しました）
3. dis850　（Ver.2.70 で修正しました）
4. dump850　（Ver.2.70 で修正しました）
5. hx850　（Ver.2.50 で修正しました）
6. romp850　（Ver.2.60 で修正しました）

【回避策】 コマンドファイル内で多バイト文字／半角カナ文字は使用しないでください。プロジェクト・マネージャ使用時は，オプションはコマンドファイルで指定されますので，オプションに多バイト文字は使用しないでください。

【改善策】 Ver.2.70 で修正しました。

No. 62　構造体パッキング時の構造体ポインタの制限

【内　容】 構造体パッキング機能(-Xpack)を使用して，

- 構造体パッキングされる構造体ポインタを register 宣言している。
- 最適化オプションに -Os, または，-Ot を指定して，構造体パッキングされる構造体ポインタがレジスタ割り付けとなる。

上記のいずれかに該当する場合，不正なコードとなる場合があります。

（例）
```
struct test {
    struct test *next ;
} ;

struct test  list[20];
struct test  *Head = &list[0] ;

func() {
    int i;
    struct test *pp ;

    pp = Head ;
    for (i=0; pp!=0; i++) {
        pp = pp->next;    /* この行に対するコードが不正となる */
    }
}
```

【回避策】 構造体ポインタの register 宣言の削除，および，最適化指定を行なわないことで回避して下さい。

【改善策】 Ver.2.41 で修正しました。

No. 63　構造体への代入，参照の制限

【内　容】 デバイス品種として V850E を指定し，最適化オプションに“ -Os -Ol ”を指定している場合，unsigned short 型のメンバをもつ構造体のメンバへの代入，参照で不正な最適化を行い，不正にコードが削除されてしまう場合があります。

（例 1）

```
unsigned int  Testflg = 0;

 struct test {
    unsigned char   b1:3;
    unsigned short  s1:2;
    unsigned short  s2:2;
    unsigned char   b2:2;
    unsigned long   l:2;
 } ;

 void func(struct test arg1)
 {
    struct  test ausrt1 = arg1 ;

    ausrt1.b1 = 2;
    ausrt1.s1 = 2;
    ausrt1.b2 = 2;    /* この行に対するコードが不正に削除される */
    ausrt1.s2 =  2;
    ausrt1.l  =  2;

    if (ausrt1.b2  != 2) {
        Testflg++ ;
    }
 }

 main() {
    struct test  arg;
    func(arg);
 }
```

該当部分に対して，下記のようなアセンブル・コードを出力しますが，

```
 #  : ausrt1.b2 = 2;
          :
st.b  r10, -3+.A4[sp]  … (1)
#  : ausrt1.s2 = 2;
ld.hu  -4+.A4[sp], r15 … (2)
mov    0x40, r16
and    0xffffff9f, r15
or     r16, r15
st.h   r15, -4+.A4[sp] … (3)
```

※ メモリ・イメージ

```
-4+.A4[sp]  -3+.A4[sp]
    ↓           ↓
 ┌─────────┬─────────┐
 │         │         │
 └─────────┴─────────┘
           <-----> st.b -3+.A4[sp]
 <---------------> ld.hu 4+.A4[sp]
```

（1）の st 命令で書き込むメモリ・エリアに対して，（2）の ld 命令で読み込むメモリ・エリアが，（1）のメモリ・エリアの一部を読み込むようなコードが連続した場合，（2）に対しての，メモリ・エリアの使用状態のチェックが不正に行われていたため，（1）のコードを不正に削除してしまう。

（例 2）

```
union test {
    char c[4];
    unsigned short us;
    unsigned int ui;
};

void func() {
    union test test;

    test.c[1]= test.c[0] = 2; /*この行に対するコードが不正に削除される*/
    test.ui  = ((unsigned)test.us & (unsigned)0xffffe7ff) | 0x1000;
    func3(test);
}
```

（例 1）と同様，（1）のコードを不正に削除してしまう。

```
# :  test.c[1] = test.c[2] = 2;
mov   2, r10
st.b  r10, -6+.A4[sp]
st.b  r10, -7+.A4[sp] ・・・ (1)
# :  test.ui = ((unsigned)test.us & (unsigned)0xffffe7ff) | 0x10000;
ld.hu  -8+.A4[sp], r14
and   0xffffe7ff, r14
or    0x10000, r14
st.w  r14, -8+.A4[sp]
```

【回避策】 -Wi,+stld_trans_opt=OFF を指定して，不具合に該当する最適化機能の抑止，または，最適化オプション“ -Ol ”の指定を外して回避して下さい。

【改善策】 Ver.2.41 で修正しました。

No. 64　関数間を跨いだ最適化処理の制限

【内　容】 最適化オプションに，-Os，または，-Ot を指定している場合，以下の 5 つの条件が同時に成り立つ場合に，関数間最適化で不正な最適化を行い，不正にコードが削除されてしまう場合があります。

1. 最初に呼び出す関数（f3）の中で，呼び出す関数（f2）の引数がアドレス渡し（ポインタ）である場合
2. 最初に呼び出す関数（f3）の中で，ポインタにアドレス代入する変数の値を，呼び出す関数（f2）の前で代入を行なう場合
3. 次に呼ばれた関数（f2）の中で，呼び出す関数（f1）の引数が値渡しとなる場合
4. 次に呼ばれた関数（f2）の中で，呼び出す関数の直前に，ソース行もしくは，ソース行の一部が，最適化（広域最適化／ループ最適化等の最適化処理）により移動した場合
5. 関数（f2）の中で，関数（f1）引数の参照が関数（f1）の呼び出しの後にない場合

（例）
```
int f1(int i){
    i++;
    return i;
}

void f2(int* p, int i){
    int j = 0                        ・・・ 4
    if(i == 0){
        volatile int vi = f1(*p);    ・・・ 3
        for(; j < 255; j++)
                ;
    }
}

void f3(void){
    int i = 1;          /* この代入コードが不正に削除される */
    int* p = &i;                     ・・・ 2
    f2(p, 0);                        ・・・ 1
}
```

A) 1～3 は，不具合発生条件の 1～3 に該当します。
B) 4 のコードが広域最適化処理により，3 の直前に移動されるため，4 の条件に該当します。4 のコードが下の if 文内でしか使われないため，if 文内の 3 の前に最適化により移動する。
C) 以降の部分で，関数引数の参照がなく，5 の条件に該当します。

【回避策】 -Wp,-f- -Wo,-Ni を指定して，不具合に該当する最適化機能の抑止，または，最適化指定を行なわないことで回避して下さい。

【改善策】 Ver.2.41 で修正しました。

No. 65　unsigned char 型変数の tidata_word セクションへの配置の制限

【内　容】　デバイス品種として V850E を指定し，unsigned char 型の変数を tidata_word セクションに配置した場合，不正コードになってリンクエラーとなる場合があります。
リンクエラーとならない場合は，本不具合に該当しません。

（例）
```
#pragma section tidata_word begin
char dummy[128];
struct test {
    unsigned char  a;
    unsigned short b;
    unsigned int   c;
} test_u;
#pragma section tidata_word end

unsigned char ua;

void func() {
    ua = test_u.a * ua;   /* この行に対するコードが不正となる */
}
```

【回避策】　unsigned char 型の該当変数を tidata_byte セクションに配置してください。

【改善策】　Ver.2.41 で修正しました。

No. 66　前処理命令での定数式の制限

【内　容】　#if 前処理命令に乗算を含む定数式を記述した場合，不正な処理となります。

（例）
```
#define A 1
#define B 2

void main(){
#if (A * B) == 2
    func1();
#else
    func2();  /* 条件判定が不正に処理され，こちらのコードが出力される */
#endif
}
```

【回避策】　ありません。

【改善策】　Ver.2.41 で修正しました。

No. 67　アセンブラ最適化処理でのコード移動の制限

【内　容】　下記，4 点の全ての条件に当てはまる場合，

1. デバイス品種として V850 コアのデバイス指定。ただし，アセンブラ・オプションで“ V850 共通オブジェクト・ファイル生成オプション-cn ”を指定している場合は，V850E を含め全てのデバイス品種指定で該当
2. 最適化オプションに“ -Os ”または“ -Ot ”を指定している。またはアセンブラ・オプションで最適化オプション“ -O ”を指定している。
3. デバッグ情報出力オプション“ –g ”を指定していない。
4. CPU 不具合回避オプション（コンパイラ・オプション-Xv850patch，または，アセンブラ・オプション“ -p ”）を指定していない。

アセンブラの最適化処理により，スタック・ポインタの書き換えコードが移動し，割り込み処理の起こるタイミングによって動作が不正になる場合があります。

（例）

・ アセンブラ・コード

```
addi  4, sp, r15
ld.w  0x4[r15], r16
st.w  r16, 0x0[r14]
ld.w  0x8[r15], r17
add  8, sp         -- スタック・ポインタ書き換えコード
jmp [lp]
```

・ アセンブラ最適化後のアセンブル・コード

```
addi  4, sp, r15
ld.w  0x4[r15], r16
add  8, sp              -- ここにコードが移動    ・・・ 1
st.w  r16, 0x0[r14]     --                      ・・・ 2
ld.w  0x8[r15], r17     --                      ・・・ 3
jmp  [lp]
```

1～2，または，2～3 の間で割り込み処理が発生すると，その後の動作が不正になります。

【回避策】　コンパイラ・オプションで“ +Oa- ”を指定するか，アセンブラ・オプションで“ +O ”を指定して回避して下さい。

【改善策】　Ver.2.41 で修正しました。

No. 68　最適化処理時の異常終了の制限

【内　容】　最適化オプションに“-Os”または“-Ot”を指定している場合，以下の 3 点に該当する関数記述のあるソース・ファイルで，コンパイルの最適化処理中に異常終了します。

1. アドレスへの直接アクセスのみ
2. 構造体以外の引数を持たない
3. 戻り値がない

（例）
```
#define ADDRESS ((unsigned char*)0x100000)
void func(void)
{
    *ADDRESS = ((*ADDRESS >> 4) & 0x0f) | ((*ADDRESS << 4) & 0xf0);
}
```

【回避策】　-Wo,-Nj を指定して，不具合に該当する最適化機能の抑止，または，最適化指定を行なわないことで回避して下さい。

【改善策】　Ver.2.41 で修正しました。

No. 69　プロジェクト・マネージャのビルド時の制限

【内　容】　ビルド実行時にプロジェクト・マネージャが無反応になり，一切の操作を受け付けなくなる場合があります。

【回避策】　ありません。
プロジェクト・マネージャが無反応になってしまった場合は，プログラムを強制終了し，プロジェクト・マネージャを再起動して下さい。

【改善策】　Ver.2.41 で修正しました。

No. 70　#pragma section 内の static 関数記述の制限

【内　容】　#pragma section 内に static 関数のプロトタイプ宣言と定義を記述した場合,次のエラーとなります。

（メッセージ）
E2211: redeclaration of 関数名

（例）
```
#pragma section ... begin
static void func();
static void func(){        /* 二重定義エラーとなる */
        :
}
#pragma section ... end
```

【回避策】　#pragma section 指定の外で，static 関数のプロトタイプ宣言や定義を行ってください。

【改善策】　Ver.2.41 で修正しました。

No. 71　オフセット付きアドレスへのアクセスでのポインタ型定数の制限

【内　容】 コンパイラの基本最適化により，式の変形による最適化処理において，オフセット付きのアドレスへの（間接）アクセスでポインタ型定数の値がポインタ値として 24 ビットを越えるような場合に，不正なコードとなります。

（例）
```
void func(int adr)
{
    *((volatile unsigned char *)0x4002000 + adr) = 0xff;
}
```

式の変形処理

1. **オフセット付きのアドレスへのアクセスの場合，アドレス**(0x4002000) ＋ **オフセット**(adr) **でのアドレス計算式にする。**
2. **この式を基本最適化処理によって“アドレス［オフセット］”の形でのアクセス式に変形する。**
3. 1 **のアドレス計算のため，アドレス値（**0x4002000**）を格納していた領域から，**2 **の式変形時に，そのアドレス値を別の領域に格納する。**

3 **での処理で，もとの領域よりサイズの小さい領域へアドレス値を格納してしまっていたため，収まらない部分が欠落してしまう。**

（出力コード例）
```
mov   255, r12
st.b  r12, 8192[r11]
```
この値が不正となる（8192）

【回避策】 **あらかじめアドレス計算を行ってから（オフセットが付かない形にしてから）値の代入を行なうように，ソース記述の変更により回避してください。**

（回避記述例）
```
void func (int adr) {
    *((volatile unsigned char *)(0x4002000 + adr)) = 0xff;
}
```

【改善策】 Ver.2.50 **で修正しました。**
ただし Ver.2.41 **では，本制限に該当する場合，以下のエラー・メッセージを出力してコンパイルを中止します。**
（メッセージ）

E2179: compiler restriction: offset out of range [24bit]

No. 72　if 文の最適化処理の制限

【内　容】 最適化オプションに，オプショナル最適化(−OI)，または，強力な最適化(−Wi,−O4)を指定している場合，if 文の条件式にビット・フィールドが関係し，その条件式の分岐条件がコンパイル時に決定できる場合，不正なコードとなる場合があります。

（例）
```
void func()
{
    struct st {
        int x:30;
    } st;

    int k = 1;
    int st.x = 80;

    k=(st.x==50); if(k!=0) f(1);
    k=(st.x!=50); if(k!=1) f(2);
    k=(st.x!=70); if(k!=1) f(3); /* f(3)を呼び出すコードが出力される */
    k=(st.x!=70); if(k==0) f(4);
}
```

上記の例では，if 文の全ての条件が成り立たないので，最適化により f(x)を呼び出すコードが全て削除可能であるが，不正な最適化処理により f(3)を呼び出すコードを出力してしまう。

【回避策】 -Wi,+data_flow_opt=0 を指定して，不具合に該当する最適化機能の抑止，または，オプショナル最適化（-OI），および，強力な最適化（-Wi,-O4）の指定を外して回避して下さい。

【改善策】 Ver.2.41 で修正しました。

No. 73　switch 文の最適化の制限

【内　容】　次の条件をすべて満たす場合に，本来であれば最適化によりレジスタを割り付けない変数を不正に割り付けてしまい，不正なコードが出力される可能性があります。

1. 最適化を指定している（-Os，-Ot オプション）
2. switch 文がテーブル分岐形式のコードになる
   - -Xcase=table を指定している
   - -Xcase=ifelse，-Xcase=binary を指定していなく，次の条件を同時に満たす場合
     - case ラベルの個数が 4 個以上
     - ラベルの値の下限と上限の差が case の数の 3 倍まで
3. switch 文記述が，次のいずれかの条件を満たす
   - case ラベルの値が連続した数値ではない（詰まっていない）。ただし case ラベルの記述の順番は関係ありません。
   - 複数の case ラベルが同じ場所に分岐している
   - break 等を記述せずに case ラベルの処理が続けて処理される

（例）　最適化オプション-Ot を指定しコンパイルした場合 ≪条件 1≫

```
void    g(void);
int     i;
void f(void){
    i = 0;
    do {
        switch(i){
            case 0:         ← case ラベルの数 4 個，下限と上限の差が 3 のため
                i++;          テーブル分岐形式のコードになる ≪条件 2≫
                break;
            case 1:
                g();
                i++;        ← break を記述せずに case ラベルの処理が続けて
            case 2:           処理される ≪条件 3≫
                i++;
            case 3:
                g();
                i++;
                break;
            default:
                g();
                break;
          }
      } while(i < 4);
  }
```

【回避策】　次のいずれかの方法で回避してください。

1. 最適化オプションの指定を削除し，デフォルトの最適化にする
2. -Xcase=ifelse を指定する（CA850 Ver.2.40 以上）
3. -Xcase=binary を指定する（CA850 Ver.2.40 以上）

【改善策】　Ver.2.50 で修正しました。

No. 74　ビット・フィールドの初期化の制限

【内　容】　ビット・フィールドのビットが“ 32 ”と指定され初期化した場合に，その値が不正になります。
また，初期化以外の場合には，不正なコードは出力されませんが，次のメッセージを出力します。

（メッセージ）

W2172: constant out of range

（例）
```
int func() {
    struct {
    unsigned int x:32;     /* 32 ビットのビット・フィールド */
    } s1={0x12345678};     /* 初期化 */
}
```

【回避策】　ビットで 32 の指定を，ビットではなく 32 ビットの型にしてください。

（例）
```
int func(){
    struct {
        unsigned int x;
    } s1={0x12345678};
}
```

【改善策】　Ver.2.50 で修正しました。

No. 75　自動変数配列の初期化の制限

【内　容】　次の条件をすべて満たす場合に，自動変数配列の初期化で不正なコードが出力されます。

1. 最適化を指定している（-Os，-Ot オプション）
2. 自動変数配列を初期化する
3. 2 の自動変数配列の初期値を 32 個以上省略する

（例）
```
int a[37]={1, 2, 3, 4, 5};   /* 6 個目以降（32 個）が省略されている */
```

【回避策】　次のいずれかの方法で回避してください。

1. 最適化オプションの指定を削除しデフォルトの最適化にする
2. 省略する初期化子に 0 を指定して，省略する部分を 31 個以内にする

（例）
```
int a[37]={1, 2, 3, 4, 5, 0}; /* 7 個目以降（31 個）が省略されている */
```

【改善策】　Ver.2.50 で修正しました。

No. 76 機種依存最適化部の ov フラグの制限

【内　容】 次のすべての条件を満たす場合に，機種依存最適化部で不正なコードを出力します。

1. オプショナル最適化（-Ol），または，強力な最適化（-Wi,-O4）を指定
2. データフロー最適化時に，機種依存最適化部で，次の演算（add，addi，sub，subr，cmp）が存在する。
   - 非負数同士の加算が桁あふれで負数になる
     （例）`0x7fffffff + 0x1 = 0x80000000`
   - 負数同士の加算が桁あふれで非負数になる
     （例）`0x80000000 + 0x80000000 = 0x0`
   - 非負数から負数を減じて負数になる（比較も同様）
     （例）`0x7fffffff - 0x80000000 = 0xffffffff`
3. 2 の ov フラグを参照する条件分岐／条件付代入（<，<=，>，>=）を行なう
4. 2，3 の 2 つの条件が広域最適化部では検出されてない

【回避策】 次のいずれかの方法で回避してください。

1. -Wi,+data_flow_opt=0 を指定して，不具合に該当する最適化機能の抑止してください。本オプションは，データフロー最適化を抑止するオプションです。
2. オプショナル最適化（-Ol），および，強力な最適化（-Wi,-O4）の指定をはずして回避してください。

【改善策】 Ver.2.50 で修正しました。

No. 77 機種依存最適化部のシフトの制限

【内　容】 次の条件をすべて満たす場合に，算術右シフトした値同士の等号比較を，論理右シフトに変換してしまう可能性があります。

1. 最適化オプション（-Os，-Ot）を指定
2. 2 つの値を算術右シフトする
3. それぞれのシフト量が異なる
4. 2 つの値が，それ以降参照されない。
5. 2 つの値が，ともに負数である。

（例） 下記で，`a=-2, b=-4`（2 つの値が負）の時に該当します

```
int func(int a, int b) {
    a >>= 1;   /* 算術右シフト */
    b >>= 2;   /* 算術右シフト */
    if (a==b) return 1;   /* 等号比較 */
    else  return 0;
}
}
```

【回避策】 次のいずれかの方法で回避してください。

1. -Wi,+hole4_opt=0 を指定して，不具合に該当する最適化機能の抑止してください。本オプションは，4 命令のピープホール最適化を抑止するオプションです。
2. 最適化オプション（-Os，-Ot）の指定をはずしてください。

【改善策】 Ver.2.50 で修正しました。

No. 78 整数とポインタの共用体の制限

【内　容】 整数とポインタの共用体で，整数の代入後のポインタアクセスで，コンパイル・エラーになってしまいます。

C5102:internal: internal error in 'IvarNode::IvarNode()'

該当する条件は，次のすべてを満たす場合です。

1. 最適化オプション（-Os，-Ot）を指定している
2. int(long)，または，unsigned int(long)とポインタの共用体がある
3. 2 の共用体の整数に整数定数を代入する
4. 代入後，2 の共用体のポインタを用いて間接参照，間接代入，構造体代入のいずれかを行なう
5. 3 の整数定数が，4 のポインタアクセスの位置まで基本ブロック内の複写の伝播で最適化できる

（例）

```
struct S {
    int a[10];
} s;
union {
    int  i;
    int* p;
    struct S* sp;
} u;
volatile int vi;
void f(void){
    u.i = 0x100;    /* 間接参照 */
    vi = *u.p;
    u.i = 0x200;    /* 間接代入 */
    *u.p = vi;
    u.i = 0x300;    /* 構造体代入 */
    s = *u.sp;
}
```

【回避策】 ポインタではなく，アドレスをそのまま記述してください。

（例）

```
struct S {
    int a[10];
} s;
union {
    int  i;
    int* p;
    struct S* sp;
} u;
volatile int vi;
void f(void){
    u.i = 0x100;        /* 間接参照 */
    vi = *(int*)0x100;
    u.i = 0x200;        /* 間接代入 */
    *(int*)0x200 = vi;
    u.i = 0x300;        /* 構造体代入 */
    s = *(struct S*)0x300;
}
```

【改善策】 Ver.2.50 で修正しました。

No. 79　構造体パッキング指定時のビット・フィールド・メンバへの複合代入の制限

【内　容】 構造体パッキング指定時にビット・フィールド・メンバの複合代入において，次のすべての条件を満たす場合に不正な値となってしまいます。

1. 複合代入の対象となるビット・フィールド・メンバが 32 ビットをまたぐ場合
2. 構造体配列で要素が解決できない

（例） パッキング値 1（-Xpack=1）でコンパイルした場合

```
struct S{
    short s1:7;
    short s2:7;
    short s3:7;
    short s4:7;
    short s5:7;    /* 32 ビット境界をまたぐ */
    short s6:7;
}sobj[10];

int i = 1;        /* 変数 */

void f1(){
    sobj[i].s1 = 1;
    sobj[i].s5 = 1;
    sobj[i].s6 = 1;
    sobj[i].s1 += 1;
    sobj[i].s5 += 1;   /* 複合代入 */
    sobj[i].s6 += 1;
}
```

（注意） 次の場合には，コンパイル時に要素が解決されるので該当しません。

```
struct S2{
    short s1:7;
    short s2:7;
    short s3:7;
    short s4:7;
    short s5:7;
    short s6:7;
}sobj2[10];

const int i = 5;  /* const: 定数 */

void f2(){
    sobj2[i].s1 = 1;
    sobj2[i].s5 = 1;
    sobj2[i].s6 = 1;
    sobj2[i].s1 += 1;
    sobj2[i].s5 += 1;
    sobj2[i].s6 += 1;
}
```

【回避策】 **次のいずれかの方法で回避してください。**

1. **複合代入を単純代入にする**

（例）
```
sobj[i].s5 = sobj[i].s5 + 1;
```

2. **ビット・フィールド・メンバが 32 ビットをまたがないように指定する**

（例）
```
struct S{
    short s1:7;
    short s2:7;
    short s3:7;
    short s4:7;
    short dummy:4;    /* ダミーのメンバ */
    short s5:7;       /* 32 ビット境界をまたがない */
    short s6:7;
}sobj[10];
```

【改善策】 Ver.2.50 **で修正しました。**

No. 80　整数とポインタの共用体の最適化に関する制限

【内　容】　次の条件をすべて満たす場合に，最適化部がエラー・メッセージを出力せずにエラー終了してしまう場合があります。エラー終了せずに正常終了した場合には，本制限事項には該当しません。

1. 最適化を指定している
   -Og，または，-O，または，-Os，または，-Ot オプション（Ver.2.50）
   -Os，または，-Ot オプション（Ver.2.41 以前）
2. ポインタと 4 バイト整数の共用体が存在する。
3. 2 の共用体で整数からポインタへの " 基本ブロック内の複写の伝播 " の最適化が起こる。
4. 3 の最適化で伝播したものが整数定数以外である。
5. 伝播先のポインタで間接アクセスするか構造体転送を行なう。

（例）　最適化オプション-Os を指定しコンパイルした場合 ≪条件 1≫

```
typedef union {
    unsigned char *p;
    unsigned long  a;
}TEST;

int e;

char
func(unsigned long data){
     register TEST  s;

    s.a = data;
    if (*(s.p) > 0x00){
        if (*(s.p) != 0xFF){
            e++;
        }
    }
    return(1);
}
```

- ポインタと 4 バイト整数の共用体 ≪条件 2≫ （unsigned char *p; を指す）
- 変数 data が変数 s.p に複写の伝播される ≪条件 3≫ （s.a = data; を指す）
- 変数 data は整数定数ではない ≪条件 4≫ （s.a = data; を指す）
- 変数 s.p の間接参照を行なう ≪条件 5≫ （if (*(s.p) > 0x00){ を指す）

【回避策】　次のいずれかの方法で回避してください。

1. 最適化レベルを下げる
   -Od，または，-Ob（CA850 Ver.2.50）
   デフォルトの最適化（CA850 Ver.2.41 以前）
2. -Wo,-Nc オプションを指定する
   基本ブロック内の複写の伝播を抑止します。

【改善策】　Ver.2.60 で修正しました。

No. 81　　パッキングされた構造体引数の制限

【内　容】 次のすべての条件を満たす場合に，スタックフレームを不正に生成します。本来であれば，引数レジスタ領域がスタックの先頭に配置されるべきですが，中央に配置されます。
戻り値が構造体である場合には，下記の条件の引数は，第 1 引数が戻り値，第 2 引数が関数の第 1 引数，第 3 引数が関数の第 2 引数のように，第 1 引数に構造体の戻り値を挿入した形になりますので，注意してください。

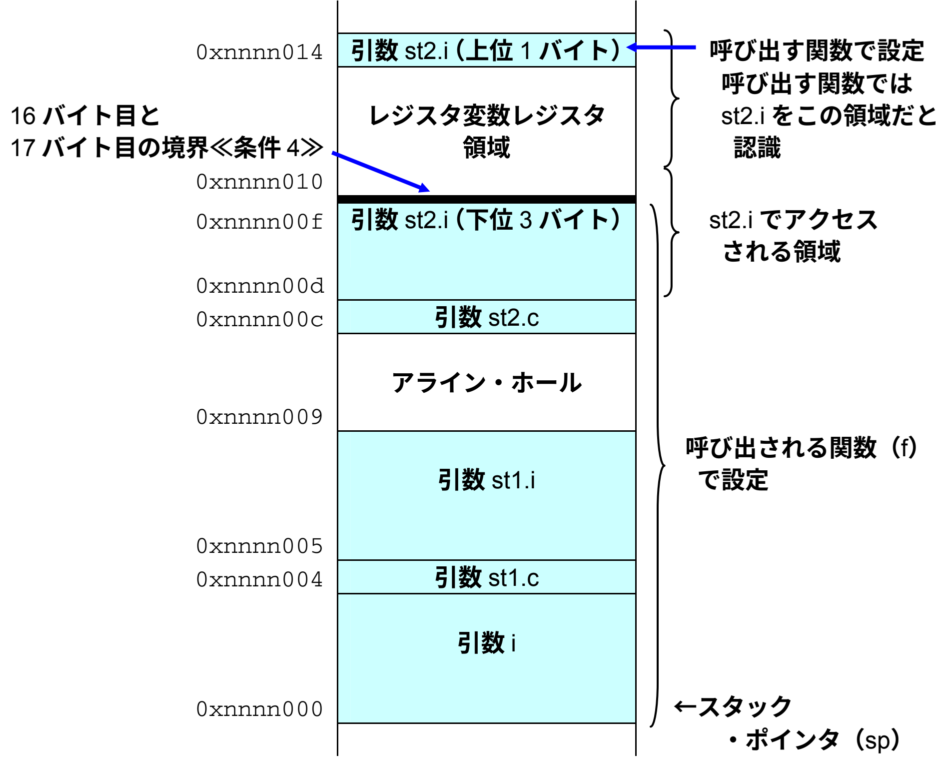
1. 構造体パッキング機能を使用している
   → -Xpack オプションを使用している，または，#pragma pack の記述がある
2. 構造体パッキングされた 5 バイト以上の構造体が関数の引数に 2 つ以上ある
3. 2 の関数が可変個引数ではない
4. 構造体として 2 つ目以降の引数になるパッキングされた構造体が，スタックフレーム内の引数領域の 16 バイト目と 17 バイト目にまたがる

そのため，条件 4 の構造体の配置が不正になり，次のアクセスに対して不正なコードを出力します。

- 構造体としてのアクセス
- スタックフレーム内の引数領域の 16 バイト目と 17 バイト目にまたがる構造体メンバのアクセス

（例）
```
#pragma pack(1)        ≪条件 1≫
typedef struct {
    char c;
    int  i;
}ST;
void f( int i, ST st1, ST st2 ) {    ≪条件 2，3≫
    g(st2.c, st2.i);
}
```

次のようにスタックに配置されます。



【回避策】 **次のいずれかの方法で回避してください。**

1. **該当する構造体の構造体パッキング機能を止める**
2. **引数を構造体へのポインタにする**

【改善策】 Ver.2.60 **で修正しました。**
また，本制限に該当するかをチェックするツールを用意しています。ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 82　データ・スワップ組み込み関数の制限

【内　容】 次の条件をすべて満たす場合に，コード生成部で次のメッセージを出力して終了してしまいます。

1. V850E のデバイスを品種指定している
2. 組み込み関数の __bsh，__bsw，__hsw のどれかを使用している
   （V850E のデバイスを指定しているときのみ使用可能な組み込み関数です）
3. 2 の引数に特定のレジスタが割り付く
4. 2 の結果を 2 箇所以上に代入している
5. 4 の代入先の 1 つ以上が，3 のレジスタと異なるレジスタである
6. 3 のレジスタを後で使おうとする

（メッセージ）
　C6202:internal: register overflow

（例 1）
```
int gi;
void func1() {
    int i1, i2;
    i1=i2=__bsw(gi);
    d_func(i1,i2);
}
```

（例 2）
```
int gi;
void func2() {
    volatile int i1, i2;
    if(i1=i2=__bsw(gi)){
        e_func(i1+i2);
    } else {
        e_func(i1-i2);
    }
}
```

【回避策】 該当する組み込み関数の戻り値を代入する変数を，volatile 指定してください。式の途中に該当する組み込み関数を記述しないでください。

（例 1）
```
int gi;
void func1() {
    volatile int i1, i2;   /* 戻り値の変数を volatile 指定 */
    i1=i2=__bsw(gi);
    d_func(i1,i2);
}
```

（例 2）
```
int gi;
void func2() {
    volatile int i1, i2;
    i1=i2=__bsw(gi);        /* 式の外に組み込み関数を移動 */
    if( i1 ){
        e_func( i1+i2 );
    } else {
        e_func( i1-i2 );
    }
}
```

また，次のいずれかの方法で，レジスタの割り付ける条件が変わるため回避できる可能性があります。

1. 最適化レベルを変える
2. 式の順序を変える，変数を挟む

【改善策】 Ver.2.50 で修正しました。

No. 83　構造体を引数とする関数のインライン展開の制限

【内　容】 次のすべての条件を満たす場合に，引数に対するコードが不正に削除されます。

1. 構造体を引数とするインライン展開対象の関数
2. 1 の関数内で使用する引数の構造体のメンバが以下の場合
   a. メンバ変数が " 配列 " または " 構造体 " または " 共用体 " で，構造体メンバの 2 番目以降に定義されている
   b. a のメンバ変数へ，配列要素を変数でアクセスする
   c. a の構造体のオブジェクトや他のメンバ変数を使用していない

なお，構造体パッキング指定時，かつ，引数である構造体が 4 バイトアラインされない場合には，上記の条件を満たしていても，この制限事項には該当しません。

（例）

```
struct S{
    int dummy1;
    int dummy2;
    int buf[1];    /* 条件 2-a */
};
struct S sobj;
int num;
static void sub(struct S sobj){  /* 条件 1 */
    int i = 0;
    num = sobj.buf[i];    /* 条件 2-b */
}                         /* 条件 2-c */
func(){
    sub(sobj);
}
```

（例のコンパイル結果）

```
_func:
    add  -.S5, sp
    ld.w  -4+.A5[sp], r11   ← 引数として値が渡されない
    st.w  r11, $_num
    #  19: }
    mov  r0, r10
    add  .S5, sp  -- 1
    jmp [lp]
```

【回避策】 次のオプションを指定して回避してください。

・ -Wp,-a-（インライン展開する関数の未参照引数の削除を抑止する）

【改善策】 Ver.2.60 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 84　関数ポインタの制限

【内　容】　次のすべての条件を満たす場合に，関数ポインタに不正な値を代入する場合があります。

1. マスクレジスタを使用している（-Xmask_reg オプションを指定している）
2. 1 コンパイル内に，下記の変数，関数のシンボルが合計 254 個以上ある（extern シンボルも含みます）
   - ・関数名
   - ・変数名
3. 関数ポインタを使用している
4. 3 の関数ポインタに，特定の関数のアドレスを代入する

（例）
```
-Xmask_reg オプションを指定 /* 条件 1 */
        :
（関数が 253 個）             /* 条件 2 */
        :
int func254() {return 254;}
        :
int (*funcp)();               /* 条件 3 */
void main() {
    funcp = &func254;         /* 条件 4 */
}
```

【回避策】　次のいずれかの方法で回避してください。

1. マスクレジスタ指定のオプション（-Xmask_reg）を削除する
2. 1 コンパイル内の変数／関数名等を 253 個以内にする

【改善策】　Ver.2.60 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 85　組み込み関数__sasf の制限

【内　容】 次のすべての条件を満たす場合に，__sasf() で不正なコードが出力されます。

1. V850E を品種指定している
2. マスクレジスタを使用している（-Xmask_reg オプションを指定している）
3. 組み込み関数__sasf()を使用している
4. __sasf() の第 1 引数が tidata／tibss 配置の unsigned short 型か unsigned char 型の変数である
5. __sasf() の第 2 引数が条件式である

（例）　V850E デバイスを指定し，-Xmask_reg を指定　/* 条件 1，2　*/

```
int gj, i1;
#pragma section tidata begin
unsigned short tus;
#pragma section tidata end
void func() {
    gj = __sasf(tus, i1 > 0);        /* 条件 3，4，5 */
}
```

（例のコンパイル結果）

```
ld.w    $_i1, r13
cmp     r0, r13              ← sasf のためにフラグ設定
sld.h   %_tus, r14
and     r21, r14             ← sasf のために設定したフラグを破壊
sasf    0xf, r14
st.w    r14, $_gj
```

【回避策】 __sasf の第１引数を volatile 指定の基本型の自動変数にしてください。

（例）

```
int gj, i1;
#pragma section tidata begin
unsigned short tus;
#pragma section tidata end
void func() {
    volatile unsigned short a=tus; /* volatile 指定の基本型の自動変数 */
    gj = __sasf(a, i1 > 0);
}
```

【改善策】 Ver.2.60 で修正しました。

No. 86　float 型から short／unsigned short 型への変換の制限

【内　容】　次の“条件 1”または“条件 2”をすべて満たす場合に，不正なコードが出力されます。

1. 条件 1
   a. float 型から short／unsigned short 型への型変換がある
   b. 型変換後の値を複数箇所で参照する
   c. b の 1 つ目の参照がメモリへの代入である
   d. b の 2 つ目以降の参照がレジスタへの代入だけで終わる
2. 条件 2
   a. float 型から short／unsigned short 型への型変換がある
   b. 型変換後の値を複数箇所で参照する
   c. b の 1 つ目の参照がメモリへの代入である
   d. b の 2 つ目以降の参照が式の演算数になる
   e. デバッグ情報出力指定オプション（-g）を指定している

（例）

```
float f = 70000.0;
short s;
short func() {
    return s = f;    /* 条件 1-a，1-b，1-c，1-d */
}
```

（例のコンパイル結果）

```
ld.w    $_f, r11
mov     r11, r6
jarl    ___trnc.sw, lp
mov     r6, r11
st.h    r11, $_s
mov     r11, r10         ←  返り値が int 型のままである
```

【回避策】　次のオプションを指定して回避してください。
・ -Wp,-Xer（同一レジスタの連続使用を抑止する）

【改善策】　Ver.2.60 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 87　　割り込み関数の制限

【内　容】　次の条件をすべて満たす場合に，割り込み関数内で不正に関数のプロローグ／エピローグのランタイム呼び出しの callt 命令が出力され，割り込み処理から reti で復帰しません。

1. V850E を品種指定している
2. 次のいずれかのオプション指定になっている
   a. -Xpro_epi_runtime=on を指定している
   b. 関数のプロローグ／エピローグのランタイム化が有効な最適化レベルである
      - -Os を指定している（Ver.2.41 以前）
      - -Ot 以外を指定している，または，最適化を指定していない（Ver.2.50）
3. 割り込み関数である（__interrupt 修飾子，または，__multi_interrupt 修飾子を指定している関数である）
4. スタックフレーム中の引数領域と自動変数領域と作業用レジスタ領域の合計が，4 バイト未満か，125 バイト以上である（アセンブルリスト上で，関数の最後に出力されている “ .R 数字 ” の値で確認できます）
5. 関数内で，コンパイラが “ r25～r29，lp（r31）” のいずれかを使用し，かつ “ r20～r24 ” は使用していない

（注意）この不具合は，-Xpro_epi_runtime=off を指定している場合には該当しません。

（例）　アセンブルリスト上での “ 引数領域 ” と “ 自動変数領域 ” と “ 作業用レジスタ領域 ” の合計の，確認関数の最後に次のようなシンボルが定義されています

```
.set  .S20, 0xd4
.set  .F20, 0xd4
.set  .A20, 0x80
.set  .T20, 0x0
.set  .P20, 0x0
.set  .R20, 0xc0  ←  ここの値
.set  .X20, 0xc0
```

【回避策】　次のオプションを指定して回避してください。

- -Xpro_epi_runtime=off

【改善策】　Ver.2.60 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 88　複雑な計算式のコード生成における制限

【内　容】 演算数の多い（最低でも 4 個の演算子を含む）複雑な計算式があり，かつ，その演算の中に，以下にあげる条件に該当した場合，

- 不正なコードが生成される
- コンパイル・エラー（E3228：illegal operand (must be register)）が出力される

のいずれかの現象が発生することがあります。

また演算数が 4 つ以下でも，最適化処理によって内部的に式が結合され，その結果，以下にあげる条件に該当した場合，上記現象が発生する場合があります。

本不具合は，コードを生成する時に，一時的に使用するレジスタ数が不足し，スタックを使用する必要が出てきたときに発生確率が高くなります。ただしコード生成処理時点のレジスタ使用状況やスタックの使用状況に依存するため，以下にあげる条件を満たしていても，本不具合が発生するとは限りません。

以下に，不具合が発生する可能性のある条件をあげます。発生条件は，使用している CPU（V850 コア，V850E／V850ES コア）と CA850 のバージョンによって変わってきます。

【 条 件 】

1. V850E／V850ES コアを使用し，CA850 Ver.2.40／Ver.2.41 を使用している場合，または，V850 コアを使用し，CA850 Ver.2.40／Ver.2.41／Ver.2.50／Ver.2.60／Ver.2.61 を使用している場合

　(a) 演算中に“構造体パッキング機能を使用した”構造体のメンバ参照があり，
　　(a-1) そのメンバが“奇数アドレスに配置された 2 バイト型の構造体メンバ”である場合…【例 1】
　　(a-2) そのメンバが“4 で割り切れないアドレスに配置された 4 バイト型の構造体メンバ”である場合 …【例 2】

　(b) 演算中に“構造体パッキング機能を使用した”構造体ポインタ変数からの参照があり，その構造体のパッキング値が 1 の場合で，2 バイト型や 4 バイト型のメンバへの参照である場合 …【例 3】

　(a-1)（a-2）（b）のいずれかに該当するとき，不具合が発生する可能性があります。

2. V850E／V850ES コアを使用し，CA850 Ver.2.50／Ver.2.60／Ver.2.61 を使用している場合

　(a) 演算中に“構造体パッキング機能を使用した”構造体のメンバ参照があり，
　　(a-1) そのメンバが“奇数アドレスに配置された 2 バイト型，または 4 バイト型の構造体メンバ”である場合…【例 1】
　　(a-2) その構造体が“tidata セクション”に配置されている場合で“奇数アドレスに配置された 2 バイト型の構造体メンバ”か“4 で割り切れないアドレスに配置された 4 バイト型の構造体メンバ”である場合 …【例 1】【例 2】

　(b) 演算中に“構造体パッキング機能を使用した”構造体ポインタ変数からの参照があり，その構造体のパッキング値が 1 の場合で，2 バイト型や 4 バイト型のメンバへの参照である場合 …【例 3】

　(c) 演算中に，組み込み関数__sasf()を使用している（__sasf 関数の演算結果を使用している）

　(a-1)（a-2）（b）（c）のいずれかに該当するとき，不具合が発生する可能性があります。

注意 “構造体”には“ビット・フィールド”も含まれます。「ビット・フィールドのメンバ」で関係してくるのは“ビット幅”ではなく“型”です。例えば“`int a:8;`”というメンバは 4 バイト型になり，“`short b:8;`”というメンバは 2 バイト型になります。

【内　容】
（続き）

【 例1 】

```
/* 奇数アドレスに配置されるメンバの例 */
#pragma pack(1)
struct {
    char pad1;
        ...
      /*
        ここまでに，先頭からの各メンバのサイズを足し合わせた合計が奇数に
        なっている
      */
    short mem; /* 奇数アドレス */
};
```

【 例2 】

```
/* 4 で割り切れないアドレスに配置された構造体メンバの例 1 */
#pragma pack(1)
struct {
    char pad1;
        ...
      /*
        ここまでに，先頭からの各メンバのサイズを足し合わせた合計が
        (4 の倍数+1) になっている
      */
    int mem; /* 奇数アドレス */
};

/* 4 で割り切れないアドレスに配置された構造体メンバの例 2 */
#pragma pack(1)
struct {
    short pad1;
    char  pad2;
        ...
      /*
        ここまでに，先頭からの各メンバのサイズを足し合わせた合計が
        (4 の倍数+3) になっている
      */
    int mem;
};
```

【 例3 】

```
/* パッキング値が 1 の場合で，“2 バイト型や 4 バイト型のメンバを，
    構造体ポインタ変数から参照する”場合の例 */

#pragma pack(1)
struct {
    char c;
    int  i;
} *pst1;

int i1, i2, i3, i4;

int func() {
    return((~i1) << ((~i2)<<(((i3)+(i4)) << (pst1->i))));
                                            /* pst1->i が該当 */
}
```

【回避策】 不具合に該当する計算式中の，

1. 構造体のメンバ
2. 組み込み関数__sasf()の演算結果

を volatile 宣言したローカル変数に一時的に格納し，そのローカル変数を計算式中で使用してください。

【 例 】

```
/* return の引数中の“st1.i”が不具合に該当していた場合 */

#pragma pack(1)
struct {
    char c;
    int  i;
} st1;

int i1, i2, i3, i4, i5;

int func() {
    return((~i1) << ((~i2)<<(((i3)+(i4)) << (i5 + st1.i))));
}
```

修正

```
/* 不具合回避のための修正 */

#pragma pack(1)
struct {
    char c;
    int  i;
} st1;

int i1, i2, i3, i4, i5;

int func() {
    volatile int  tmp = st1.i;
    return((~i1) << ((~i2)<<(((i3)+(i4)) << i5 + tmp)));
}
```

st1.i の値を volatile 宣言した変数に一旦格納し，その変数を不具合該当箇所で使用する

【改善策】 Ver.2.70 で修正しました。
また，本不具合に該当するかをチェックするツールを用意しています。
ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 89　インライン関数の引数に関する制限

【内　容】　インライン展開される関数（以下，インライン関数）が以下の条件を同時に満たす場合，不正なコードが生成されます。

- インライン関数の引数のアドレスを取得し，そのアドレスを“引数とは別の型のポインタ”にキャストして，その引数の内容を参照している
- 上記の引数は，その箇所以外では使用されていない

この場合，インライン関数が展開された箇所において，キャストされた結果のアドレスが不正になります。

プログラム内で#pragma inline 指定している関数がある場合，本不具合に該当する可能性があります。また最適化オプション“-Os”“-Ot”やインライン展開関連のオプションの組み合わせによっては関数がインライン展開され，本不具合に該当する可能性があります。

【 不具合再現コード例 】

```
#pragma inline subfunc

unsigned int s;

static void
subfunc(int i) {                  /* この関数がインライン展開される */
    s = *((unsigned int*)&i);     /* キャスト */
}

void
func(int ii) {
    subfunc(ii);
}

void
main(void) {
    int t = 2;

    func(t);
    printf("%x¥n", s);   /* 引数の内容参照 */
}
```

【回避策】　以下のいずれかの方法で回避してください。

1.　インライン展開を抑止する

（ア）　CA850 Ver.2.40／Ver.2.41 を使用している場合，（a-1）または（a-2）の方法を行なってください。

（a-1）#pragma inline 指定をはずし，かつ，最適化オプションを“-Os”“-Ot”「以外」にする

（a-2）#pragma inline 指定をはずさず（ソースに手を加えず），かつ，最適化オプションも変更しない場合，回避オプションを指定する。回避オプションは“#pragma inline 指定の有無”と“最適化オプションの組み合わせ”によって異なります。

<table>
<tr><th></th><th>#pragma inline 指定有り</th><th>#pragma inline 指定無し</th></tr>
<tr><td>最適化オプション
-Os，-Ot 以外指定</td><td rowspan="3">次の 3 つのオプションを
すべて指定
-Wp,-N0
-Wp,-G0
-Wp,-Sn</td><td>不具合に該当しません
（(a-1) に相当）</td></tr>
<tr><td>最適化オプション
-Ot 指定</td><td>次の 3 つのオプションを
すべて指定
-Wp,-N0
-Wp,-G0
-Wp,-Sn</td></tr>
<tr><td>最適化オプション
-Os 指定</td><td>-Wp,-Sn</td></tr>
</table>

【 PM，PM+におけるオプションの指定方法 】

“コンパイラ・オプションの設定→最適化 2”で“-Wp,-N0 指定”は“コードサイズ”に“0”を，“-Wp,-G0 指定”は“スタックサイズ”に“0”を指定します。“-Wp,-Sn 指定”は“コンパイラ・オプションの設定→その他→他のオプション”に“-Wp,-Sn”をそのまま記述します。

（イ）　CA850 Ver.2.50／Ver.2.60／Ver.2.61 を使用している場合，（b-1）または（b-2）の方法を行なってください。

（b-1）#pragma inline 指定をはずし，かつ，最適化オプションを“-Ot”「以外」にする

（b-2）#pragma inline 指定をはずさず（ソースに手を加えず），かつ，最適化オプションも変更しない場合，回避オプションを指定する。回避オプションは“#pragma inline 指定の有無”と“最適化オプションの組み合わせ”によって異なります。

<table>
<tr><th></th><th>#pragma inline 指定有り</th><th>#pragma inline 指定無し</th></tr>
<tr><td>最適化オプション
-Os，-Ot 以外指定</td><td rowspan="3">-Wp,-no_inline</td><td>不具合に該当しません
（(b-1) に相当）</td></tr>
<tr><td>最適化オプション
-Ot 指定</td><td>-Wp,-no_inline</td></tr>
<tr><td>最適化オプション
-Os 指定</td><td>不具合に該当しません
（(b-1) に相当）</td></tr>
</table>

【 PM，PM+におけるオプションの指定方法 】

“コンパイラ・オプションの設定→最適化の詳細設定”において，“インライン展開の制御”で“展開しない [-Wp,-no_inline]”を指定します。

2.　インライン関数の引数を，別の変数に代入してから使用する

3.　未参照引数の削除機能を抑止するオプション“-Wp,-a-”を指定する

PM，PM+の場合は，“コンパイラ・オプションの設定→その他→他のオプション” に，“-Wp,-a-”をそのまま記述します。

【改善策】 Ver.2.70 で修正しました。
また，本不具合に該当するかをチェックするツールを用意しています。
ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 90 ループ処理の最適化に関する制限

【内　容】 for ループ，while ループ，do〜while ループ，if〜goto ループにおいて，以下の条件をすべて満たすとき，ループ回数が 2 回以上であっても，1 回しかループしません。

【 条 件 】

1. CA850 Ver.2.40／Ver.2.41 を使用している場合

   (a) 最適化オプションが " -Os "" -Ot " 以上のとき

   (b) ループ変数として，unsigned int 型の変数を利用している

   (c) ループ変数の値が次のいずれかである

   - ループの初期値：0xfffe 以下
     ループの終了値：0xffff0002 以上
     ループの加算値：0xfffe 以下

   - ループの初期値：0xffff0002 以上
     ループの終了値：0xfffe 以下
     ループの減算値：0xfffe 以下

   (d) ループ変数の比較に " <，<=，>，>= " のいずれかを使用している

   (e) 次のいずれも行っていない
   - ループ内におけるループ変数への代入処理（ループ変数の加減算処理は除く）
   - ループ外からループ内への分岐
   - ループ内からループ外への分岐

2. CA850 Ver.2.50／Ver.2.60／Ver.2.61 を使用している場合

   (a) 最適化オプションが " -O "" -Os "" -Ot " のとき

   (b) ループ変数として，unsigned int 型の変数を利用している

   (c) ループ変数の値が次のいずれかである

   - ループの初期値：0x3ffffffe 以下
     ループの終了値：0xc0000002 以上
     ループの加算値：0x3ffffffe 以下

   - ループの初期値：0xc0000002 以上
     ループの終了値：0x3ffffffe 以下
     ループの減算値：0x3ffffffe 以下

   (d) ループ変数の比較に " <，<=，>，>= " のいずれかを使用している

   (e) 次のいずれも行っていない
   - ループ内におけるループ変数への代入処理（ループ変数の加減算処理は除く）
   - ループ外からループ内への分岐
   - ループ内からループ外への分岐

【内 容】

【 例 】

```
void
main(void) {
    unsigned int  i;

    for(i=0; i<0xffffff00; i+=0x1111) {
                :
              (処理)
                :
    }
}
```

【回避策】 次のいずれかを行ってください。

1. -Wo,-No を指定（0～1 回のループ展開の最適化抑止）
2. unsigned int 型のループ変数を volatile 宣言する
3. 最適化レベルを下げる
4. ループの比較を “ != ” で書き換える

【改善策】 Ver.2.70 で修正しました。
また，本不具合に該当するかをチェックするツールを用意しています。
ツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 91 if 文と goto 文によって未定義シンボルエラーが発生する制限

【内　容】 以下の条件をすべて満たすとき，下記のように，リンク時にユーザが定義していないラベルが“ 未定義エラー ” となって終了します（ラベル名はアプリケーションによって異なります）。

```
ld850: CA850 error F4452: undefined symbol.
.G38  (referenced in ".¥main.o")
```

【 条 件 】

1. 最適化オプションが “ -Ob ”（デフォルト）以上である
2. デバッグ情報生成オプション “ -g ” を指定している
3. if 文内に，同一変数に定数を代入する文が存在し，さらに else を使用せずに，if の最後で goto 文を記述している。またこれ以外の文は記述していない
4. if 文の条件で “ && ” や “ || ” を使用していない
5. if 文内における，変数に値を代入する文において，その代入する 2 つの定数が次の条件を満たす

- if 文の条件が成立した場合に代入される定数を a，成立しない場合に代入される定数を b とすると，これらの定数が “ 符号付整数の場合 ” は以下の（a）～（d）のいずれか，“ 符号なし整数の場合 ” は以下の（e）～（h）のいずれかを満たす。ただし，a，b の値が “ 浮動小数点数 ” や “ ポインタ ” だった場合は対象外となります（“ 整数ポインタの間接代入 ” は対象となります）

（a） a が 0 で，b が-32768 以上 32767 以下
（b） b-a が 2 の n 乗の値で，a が-32768 以上 32767 以下
（c） b が 0 で，a が-32768 以上 32767 以下
（d） a-b が-32768 以上 32767 以下，または 2 の n 乗で，かつ b が-32768 以上 32767 以下
（e） b が 0 で，a が 32767 以下
（f） a-b が 32767 以下，または 2 の n 乗で，かつ b が 32767 以下
（g） a が 0 で，b が 32767 以下
（h） b-a が 32767 以下，または 2 の n 乗で，かつ a が 32767 以下

【内　容】
（続き）

【 例 1 】

```
    int i;

      :
      :

    if(i){
        i = 0;              /* 条件 5 にある“a”*/
        goto label;         /* 条件 3 */
    }
    i = 1;                  /* 条件 5 にある“b”*/
label:
```

なお，明示的に goto 文を使用していない場合でも，コンパイラの最適化状況によって内部的に goto 文を出力することがあり，その際に上記条件を満たす場合も，制限事項に該当します。

【 例 2 】

```
if( num == 2 ){
    return 2;
}
if( num == 3 ){
    return 0;
}
return 0;
```

上記のコードは内部的には以下のようなコードを出力し，これが条件を満たすことになり，制限事項に該当することになります

```
unsigned char tmp;
    if( num == 2 ){
        tmp = 2;            /* 条件 5 にある“a”*/
        goto label;         /* 条件 3 */
    }
    if( num == 3 ){
        tmp = 0;            /* 条件 5 にある“b”*/
        goto label;         /* 条件 3 */
    }
    tmp = 0;
label:
    return tmp;
```

【回避策】 次の対処を行ってください。

- コンパイラ・オプション“-Wo,-NX”を指定（setf 最適化を抑止します）

【改善策】 Ver.2.72 で修正しました。
本不具合は，該当した場合はリンカでエラーが発生します。ユーザが定義していないラベルで undefined symbol エラーが発生した場合は，上記で示した回避策によって回避してください。

No. 92　フラッシュ／外付け ROM 再リンク機能におけるリンク時の制限

【内　容】　CA850 Ver.2.70 を使用し，かつ“フラッシュ／外付け ROM 再リンク機能”を使用している場合で，次の条件を満たすとき，フラッシュ・メモリ側のロード・モジュールをリンクする際に，次のようなリロケーション解決エラーが発生するか，エラーにはならずに不正コードが出力されることがあります。

```
ld850: CA850 error F4163: output section ".tibss.word" overflowed or illegal label reference for symbol "symbol" in file "file.o"(value: 0xffff9048, input section: .text, offset: 0x00000002, type: R_V850_REGWBYTE). "symbol" is allocated in section ".tibss.word" (file: a.out).
```

【 条 件 】

- フラッシュ側ロード・モジュールから，ブート側ロード・モジュールに存在する外部シンボルを，tp（テキスト・ポインタ）または ep（エレメント・ポインタ）相対参照を行っている。

“フラッシュ／外付け ROM 再リンク機能”を使用すると“ブート側のロード・モジュール”と“フラッシュ側のロード・モジュール”が作成されます。“フラッシュ側のロード・モジュール”から“ブート側のロード・モジュール”の外部シンボルを参照する際，ブート側のロード・モジュールのベース･ポインタからのオフセットを算出し，その値を用いてコードを生成しています。しかし，本不具合により，このベース・ポインタとなる tp，gp，ep の値が不正となってしまうため，tp，ep からのオフセット値も不正になってしまいます。
なお，gp からオフセット値に関しては，tp，ep とは異なった方法で算出しているため，gp 相対参照を行なうコードに関しては，本不具合には該当しません。

【回避策】　ありません。

【改善策】　Ver.2.72 で修正しました。
また，本不具合に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 93　　E2288 エラーの制限

【内　容】　下記の条件全てを満たす場合に，エラーメッセージ（E2288）を出力しますが，CA850 が不正終了してしまいます。

E2288：return type mismatch *xxxx* （*yyyy*）

【 条 件 】

1. 関数の戻り型と戻り値の型が変換できない
   関数の戻り型と戻り値型が不正になるような記述の場合に，E2288 エラーになります。
2. 構造体パッキング指定（-Xpack）
3. 関数の戻り型が構造体／共用体型

PM+ では，エラーメッセージが表示されエラーになりますが，エラーカウントが不正になります。

【 例 】

```
struct S sobj;
struct S func1(){
    return &sobj;        /* E2288エラー */
}
```

【回避策】　関数の戻り型と戻り値の型が正常になるような正しい記述にしてください。

【改善策】　Ver.3.20 で修正しました。

No. 94　セキュリティ ID とオプションバイトの制限

【内　容】　下記の条件全てを満たす場合に，次の値が 0 になります。

【 条 件 】

1. セキュリティ ID，または，オプションバイトを持つデバイスを品種指定している
2. リンカの-B オプションを指定している
3. セキュリティ ID，オプションバイトをアセンブラ・ソース・ファイルで定義していない

【 対象となる値 】

- -Xsid オプションで指定したセキュリティ ID
- -Xsid 省略時のセキュリティ ID の初期値（Ver.3.00 のみ）
  本来は 0xffffffffffffffffffff になります
- オプションバイトの初期値
  本来は，デバイスファイルに登録されている値になります

【回避策】　アセンブラ・ソース・ファイルで定義してください。
セキュリティ ID，オプションバイトが，デバイスファイルで定義されている必要があります

【 例 】

```
#---------------------------------------------------------------
#        SECURITY_ID
#---------------------------------------------------------------
 .section        "SECURITY_ID"
 .word  0xffffffff       --0-3 byte code,Address is 0x70-0x73
 .word  0xffffffff       --4-7 byte code,Address is 0x74-0x77
 .hword 0xffff           --8-9 byte code,Address is 0x78-0x79
#---------------------------------------------------------------
#        OPTION_BYTES
#---------------------------------------------------------------
 .section        "OPTION_BYTES"
 .hword 0x0000           --0-1 byte code,Address is 0x7a-0x7b
 .hword 0x0000           --2-3 byte code,Address is 0x7c-0x7d
 .hword 0x0000           --4-5 byte code,Address is 0x7e-0x7f
```

【改善策】　Ver.3.10 で修正しました。

No. 95　浮動小数点定数と汎整数型の制限

【内　容】　浮動小数点型の値を汎整数型に型変換する場合，整数部の値で表現できる範囲を signed int/signed long 型の値域として，それを越えて unsigned int/unsigned long 型に変換する場合には，コンパイル・エラーとなることがあります。

```
E2519: invalid has occurred at compile time.
```

【 例 】

```
unsigned int ui = 2147483647.0;         /* OK */
unsigned int ui = 2147483648.0;         /* エラー */
```

【回避策】　整数型にしてください。

【 例 】

```
unsigned int ui = 2147483648;
```

【改善策】　制限事項とします。

No. 96　入出力関数の標準ライブラリの入力変換の制限

【内　容】　**入出力関数の標準ライブラリの関数** printf, sprintf, vprintf, vsprintf **の入力変換で，変換指定子**"g,G"**に対して，指定した精度を**+1 **します。**

【 例 】

```
printf("%.2g", 12.3456789);
/* 12 となるべきですが，12.3 となってしまいます。*/
```

【回避策】　**ありません。**

【改善策】　**制限事項とします。**

No. 97　　機種依存最適化部の代入の制限

【内　容】 最適化指定時[注]に，基本ブロック内で同じ変数群 3 つ以上にそれぞれ同じ値を代入する if 文，または，switch 文があると，最適化により，最初に代入を行う変数の値が不定となることがあります。

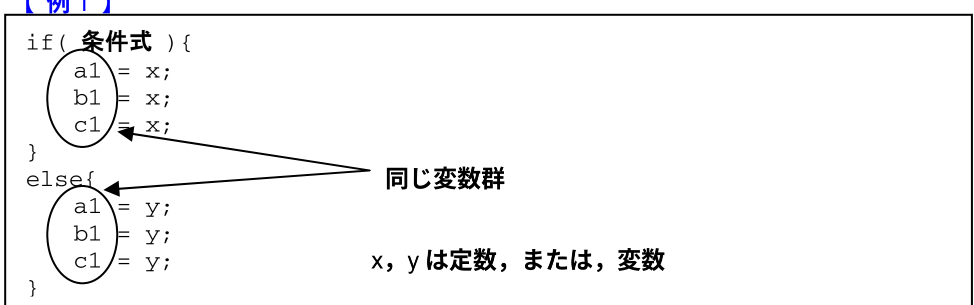
注：　Ver.2.50 未満の場合：　以下のいずれかのオプション指定の場合になります。

サイズ優先最適化＋オプショナル最適化指定（-Os -Ol）
サイズ優先最適化＋強力な最適化指定（-Os -Wi,-O4）
サイズ優先最適化＋オプショナル最適化＋強力な最適化指定（-Os -Ol -Wi,-O4）
速度優先最適化＋オプショナル最適化指定（-Ot -Ol）
速度優先最適化＋強力な最適化指定（-Ot -Wi,-O4）
速度優先最適化＋オプショナル最適化＋強力な最適化指定（-Ot -Ol -Wi,-O4）

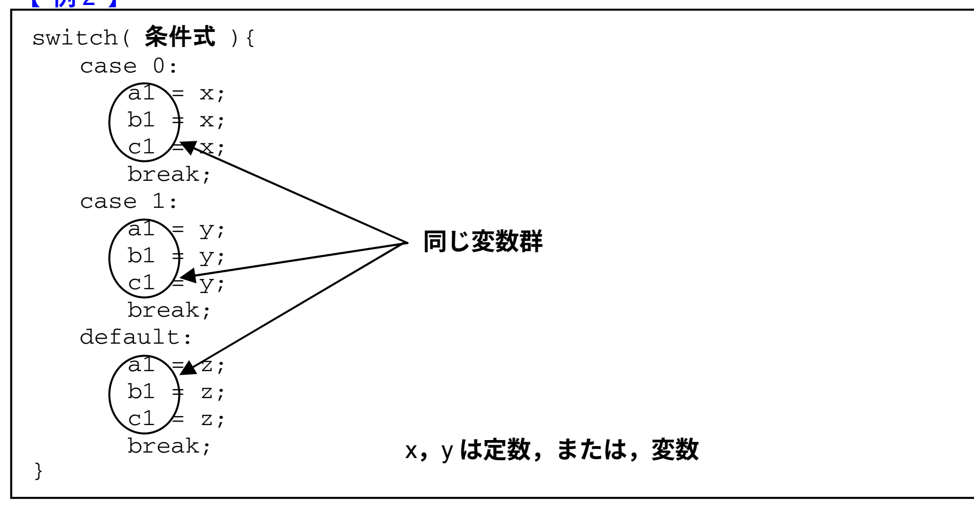
Ver.2.50 以上の場合：　以下のいずれかのオプション指定の場合になります。

サイズ優先最適化指定（-Os）
速度優先最適化指定（-Ot）
高度な最適化＋強力な最適化指定（-O -Wi,-O4）

【 例 1 】



【 例 2 】



【回避策】 次のオプションを指定し，不具合に該当する最適化機能の抑止をしてください。

-Wi,+cf_reg_trans_opt1=0
-Wi,+cf_reg_trans_opt2=0
-Wi,+cf_forward_reg_trans_opt=0
-Wi,+cf_reverse_reg_trans_opt=0
-Wi,+abs_ptn_opt=0

【改善策】 Ver.3.10 で修正しました。
また，本不具合に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせ下さい。

No. 98 自動変数領域の一部が不正に削除される制限

【内　容】 下記の条件全てを満たす場合に，自動変数領域の一部が削除され，その自動変数への間接参照が不正となる場合があります。

【 条 件 】

1. 構造体，配列，共用体型自動変数の先頭以外の直接参照またはアドレス取得をしている
2. その自動変数が，関数内の自動変数領域の先端（低位側）にある

この場合，自動変数の 1.で参照したアドレスより先頭側の間接参照が不正となります。ただし，1.で参照したアドレスより後方側の間接参照は正常に行われます。

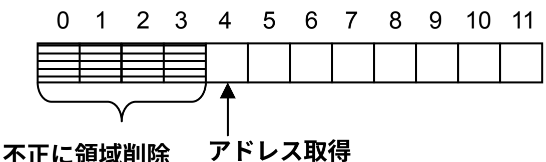
【 例 】

```
char val;
char *p;

void func(){
  char str[12];

/*ポインタの初期化として，配列 str[4]のアドレスを代入 */
  p = &str[4];
     ・
     ・
     ・
/* p を使って str[0]～str[3] を間接参照 */
  val = *(p-1);        ← この間接参照が不正
}
```

この場合， str[4] ～ str[11]までの 8 バイトの領域確保しか行われません。そのため，str[0] ～ str[3] への間接参照が不正となる場合があります。



【回避策】 volatile 修飾した外部ポインタ変数に対して， 該当する自動変数の先頭アドレスを代入してください。

【 例 】

```
char val;
char *p;
char * volatile dmy;

void func(){
  char str[12];

/* ポインタの初期化として，配列 str[4]のアドレスを代入 */
  p = &str[4];
     ・
     ・
     ・
/* volatile 修飾したダミーの外部ポインタ変数に，配列 str の先頭アドレスを代入 */
  dmy = &str[0];
/* p を使って str[0]～str[3] を間接参照 */
  val = *(p-1);        ←正常に間接参照
}
```

【改善策】 Ver.3.20 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 99　hx850 の“-U オプション”による指定アドレスの制限

【内　容】　下記の条件全てを満たす場合に，F8651 エラーとなります。

【 条 件 】

1. “-U*start,size*” または ”-U*num,start,size*” オプションを指定している
2. 1. のヘキサ変換領域指定において，周辺 I/O レジスタ領域の先頭を超える領域を指定している

```
F8651: specified address area (addr1 - addr2) overlaps I/O area (addr3 - addr4)
```

【回避策】　ありません。

【改善策】　Ver.3.20 で修正しました。

No. 100　strcmp() と strncmp() の動作が異なる制限

【内　容】　文字列を比較する際に，文字を次のように扱います。

strcmp() 関数　：符号付
strncmp() 関数：符号なし

そのため，strcmp() と strncmp() の動作が異なります。ただし，アスキー文字コードの範囲内(0x00～0x7f) の場合には，本制限は該当しません。

【 例 】

```
strcmp("aaa", "aa¥x80");
  /* "0x80"を負数として扱うため，"a"よりも"0x80"の方が小さいものとして判断します */
  /* その結果，戻り値として0より大きい整数(上記の場合"225")を返します。 */

strncmp("aaa", "aa¥x80", 3);
  /* "0x80"を正数として扱うため，"a"よりも"0x80"の方が大きいものとして判断します */
  /* その結果，戻り値として0より小さい整数(上記の場合"-31")を返します。 */
```

【回避策】　ありません。

【改善策】　Ver.3.20 で，strcmp() 関数を符号なしとして比較するように修正しました。

No. 101　65 文字以上の文字列定数の制限

【内　容】　65 文字以上の文字列定数の 2 文字目が"¥¥"の場合，次のアセンブルエラーとなります。

```
E3249: illegal syntax
```

ここでの文字列中の文字数の数え方は，文字列定数の末尾に"¥0"を付加し，エスケープ文字を 2 文字として扱います。そのため，"¥¥"や"¥0"は 2 文字となります。従いまして，以下の例は 65 文字以上の文字列定数となりますので，アセンブルエラーとなります。

【 例 】

```
char a[]="a¥¥aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
  /* E3249エラー */

【文字列の数え方】
a¥¥aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa¥0
      ↑         ↑         ↑         ↑         ↑         ↑    ↑
    10文字目  20文字目  30文字目  40文字目  50文字目  60文字目 65文字目
```

【回避策】　次のいずれかの方法で回避してください。

文字列を
"a¥¥aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
とした場合，

1. 配列の宣言時には'¥¥'を別の文字で置き換えておき，実行時に'¥¥'に書き換える。

【 例 】

```
char a[]="axaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    void main() {
      a[1] = '¥¥';
    }
```

2. char型配列を文字で初期化する。

【 例 】

```
char a[] = {   'a','¥¥','a','a','a','a','a','a','a',
               'a','a','a','a','a','a','a','a','a','a',
               'a','a','a','a','a','a','a','a','a','a',
               'a','a','a','a','a','a','a','a','a','a',
               'a','a','a','a','a','a','a','a','a','a',
               'a','a','a','a','a','a','a','a','a','a',
               'a','a','a','¥0' };
```

3. 該当文字列をアセンブラで記述する。

【 例 】

```
  .sdata
  .globl _a, 62
_a:
  .str "a¥¥aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa¥0"
```

【改善策】　Ver.3.20 で修正しました。

No. 102　アセンブラ最適化の制限

【内　容】 アセンブラ記述による擬似命令.option volatile あるいは.option volatile と同じ機能を持つ.option nooptimize により，アセンブラ最適化が行われない範囲内に存在する命令が，アセンブラ最適化の影響を受けて，命令展開により新たに生成された命令の並べ替えが行われる可能性があります。

C 言語で volatile 修飾した変数へのアクセスを記述した場合は，.option volatile 擬似命令と変数へのアクセス命令が出力されます。このときアセンブラ最適化の影響を受けて，命令展開により新たに生成された命令の並べ替えが行われる可能性があります。ただし，メモリアクセス命令の削除および順序の入れ替えは行いません。従いまして，C ソース・ファイルに対しては，ANSI-C における volatile 修飾の仕様の範囲内で正常動作します。

【 条 件 】

以下の条件を全て満たすときに，命令の出力順序が変わる可能性があります。

1. C 言語で記述している場合
   (a) 最適化オプション-Og/-O/-Os/-Ot のいずれかを指定
   (b) 別ファイルに volatile 修飾した変数が定義
   (c) 以下のいずれかの条件を満たす変数
      (ア) 配置指定していない const 変数
      (-Xsconst[=num]オプション指定、セクション・ファイルでの指定、
      #pragma section 指定を全て行っていない const 変数)
      (イ) 以下のいずれかの方法により配置指定している変数
      - #pragma section でセクション種別に data/const セクションを指定
      - -Gnum オプション指定で num より大きいサイズの変数を.data/.bss セクションに配置
      - セクション・ファイルでセクション種別に data/const を指定し，変数名を記述
      - -Xsconst[=num]オプション指定で num より大きい変数を.const セクションに配置
   (d) (b)および(c)に該当する変数へのアクセスを記述

この場合，(d)のアクセスコードが該当します。

2. アセンブリ言語で記述している場合
   (a) 最適化オプション-O を指定
   (b) 命令展開が行われるアセンブリ言語命令を.option volatile あるいは.option nooptimize 範囲内に記述
   (c) (b)に該当する命令中のラベルがその時点(行)で未定義

この場合，(b)のコードが該当します。

【内　容】
（続き）

【 例 】

```
.bss
.globl    _Label1, 1
.lcomm    _Label1, 1, 1       ← _Label1を同一ファイル内で定義

.option data  _Label2         ← _Label2を別ファイルで定義


.option volatile
st.b  r13, $_Label1
.option novolatile

.option volatile
st.b  r14, $_Label2
.option novolatile
```

この場合，以下のように_Label1，_Label2 への参照命令は 2 命令に展開されます。_Label2 への参照命令の内，命令展開により新たに生成された命令の並べ替えが行われる可能性があります。

【命令並べ替えが行われない場合】	【命令並べ替えが行われる場合】
.option volatile movhi 0x0, gp, r1 st.b r13, 0x0[r1] .option novolatile .option volatile movhi 0x0, gp, r1 st.b r14, 0x0[r1] .option novolatile	.option volatile movhi 0x0, gp, r1 st.b r13, 0x0[r1] .option novolatile movhi 0x0, gp, r1 ← この命令が移動 .option volatile st.b r14, 0x0[r1] .option novolatile

【回避策】　アセンブラ最適化を抑止してください。

1．C ソースファイルに対して
-Wa,+O オプションを指定してください。
PM+をご使用の場合は，[ツール]メニュー → [コンパイラオプションの設定] → [その他]タブの ”他のオプション” に，-Wa,+O と記述してください。

2．アセンブラソースに対して
-O オプションを指定しないようにしてください。
PM+をご使用の場合は，[ツール]メニュー → [アセンブラオプションの設定] → “最適化を行う” のチェックを外してください。

【改善策】　Ver.3.20 で修正しました。

No. 103　アドレスを取られた自動変数への代入文が不正に削除される制限

【内　容】　下記の条件全てを満たす場合に，アドレスを取られた変数への代入文が不正に削除される場合があります。

【 条 件 】

1. CA850 Ver.2.50 未満を使用している場合
   (a) 最適化オプションが-Os または-Ot である
   (b) 下記のいずれか一方を行っている
      - 自動変数のアドレスを 32bit 整数型の外部変数に代入後，関数呼び出し【例 1】
      - 自動変数のアドレスを静的に特定できない外部変数に代入後，関数呼び出し【例 2】
2. CA850 Ver.2.50 以上を使用している場合
   (a) 最適化オプションが-Og 以上である
   (b) 下記のいずれか一方を行っている
      - 自動変数のアドレスを 32bit 整数型の外部変数に代入後，関数呼び出し【例 1】
      - 自動変数のアドレスを静的に特定できない外部変数に代入後，関数呼び出し【例 2】

【 例 1 】

```
int a;

void func1(void){
  int val;
  a = &val;
  val = 1;              ← 不正に削除
  func2();
```

【 例 2 】

```
int* ary[10];

void func1(int i){
  int val;
  ary[i] = &val;
  val = 1;              ← 不正に削除
  func2();
```

【回避策】　次のいずれかの方法で回避してください。

1. オプションで回避する場合
   (a) CA850 Ver.2.50 未満を使用している場合
       最適化オプション-Os あるいは-Ot を指定しないでください。
   (b) CA850 Ver.2.50 以上を使用している場合
       最適化オプションを-Ob / -Od のいずれかに指定してください。

2. ソースの記述で回避する場合
   アドレスを取られる自動変数に対して，volatile 修飾をしてください。

【 例 】

```
  volatile int val;
```

【改善策】　Ver.3.20 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 104　ループの実行回数が不正になる制限

【内　容】　下記の 1.～5. の条件を全て満たす場合に，ループの実行回数が不正になる場合があります。

【 条 件 】

1. 最適化オプションとして"-Og" "-O" "-Os" "-Ot"のいずれかを指定している
2. 帰納変数が volatile 未指定である
3. ループの終了条件が帰納変数と定数との比較である
4. ループ内で帰納変数に対して定数の加減算を行っている
5. ループ内で帰納変数を(ア)または(イ)のように使用している
   (ア)　a.から e.の全てを満たす【例 1】
      a.　帰納変数を配列のインデックスとして使用
      b.　a.の配列の要素が構造体・共用体・配列のいずれか
      c.　a.の配列の要素サイズが 2 のべき乗以外
      d.　a.の配列の要素サイズが 65534 バイトの範囲内
      e.　ループの終了条件として指定している 3. の定数と，a.の配列の要素サイズの積が 32bit 整数を超過
   (イ)　f.から i.の全てを満たす【例 2】
      f.　帰納変数を定数との乗算の一方のみに使用
      g.　f.の定数が 2 のべき乗以外
      h.　f.の定数が-65534～65534 の範囲内
      i.　ループの終了条件として指定している 3. の定数と，f.の定数との積が 32bit 整数を超過

(注意) 帰納変数とは，ループの終了条件を制御する変数です。

【 例 1 】

```
struct {
    int s1;
    int s2;
    int s3;            //配列 ary の要素は構造体
} ary[100];            // サイズ(12 バイト)は 2 のべき乗以外で 65534 バイトの範囲内

int i;                                  //帰納変数 i は volatile 未指定
for ( i = 0; i < INT_MAX; i ++ ){       //ループの終了条件が帰納変数と定数との比較
                                        //帰納変数 i に対して定数の加算

     ary[i].s1 = 1;                     //帰納変数 i を配列のインデックスに使用
     …
}
```

INT_MAX*12(=0x5FFFFFFF4)が 32bit 整数を超過しているため，本制限に該当する可能性があります。

【 例 2 】

```
volatile int vi;

int i = 0;                              //帰納変数 i は volatile 未指定
while ( i < INT_MAX ){                  //終了条件が帰納変数と定数との比較
  vi = i * 100;                         //帰納変数 i を定数との乗算のみに使用
  …                                     //定数は-65534～65534 の範囲内

  i++ ;                                 //帰納変数 i に対して定数の加算
}
```

INT_MAX*100(=0x31FFFFFF9C)が 32bit 整数を超過しているため，本制限に該当する可能性があります。

【回避策】　下記の 1.～3. のいずれかを適用してください。

1. ループ終了条件の定数を下記のいずれかのように変更する
   - ループ終了条件の定数と，帰納変数をインデックスとする配列の要素サイズとの積が32bit 整数を超過しない
   - ループ終了条件の定数と，帰納変数と乗算を行っている定数との積が32bit 整数を超過しない

2. 帰納変数に対して volatile 指定する

【 変更前 】

```
int i;
for ( i = 0; i < INT_MAX; i ++ ){

    ary[i].s1 = 1;
    …
}
```

【 変更後 】

```
volatile int i;
for ( i = 0; i < INT_MAX; i ++ ){

    ary[i].s1 = 1;
    …
}
```

3. 最適化オプションとして"-Od" "-Ob" のいずれかを指定する

【改善策】　Ver.3.40 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 105　文字列定数の内容が不正になる制限

【内　容】　下記の 1. ～ 2. の条件を全て満たす場合，文字列定数の内容が不正になります。

【 条 件 】

1. 文字列定数中に ASCII コードの 0x00 を使用している
2. 1.に対して ASCII コードの 0x30～0x37 のいずれかが連続する

【 例 】ASCII コード 0x00 の直後の ASCII コードが 0x37 である場合

```
char string1[] = "¥x00¥x37";
char string2[] = "¥000¥067";    // (37)16 = (67)8
char string3[] = "¥x00" "7";    // (37)16 = '7 '
```

いずれも 0x00, 0x37, 0x00 が正常な出力ですが 0x07, 0x00 と不正な出力となります。

【回避策】　下記の 1.～2. のいずれかを適用してください。

1. 文字列定数を使用しないで初期化する

```
char string4[] = {'¥x00', '¥x37', '¥0'};
```

2. ASCII コードの 0x30～0x37 以外で初期化した後，動的に書き換える

```
char string5[] = "¥x00*";
string5[1] = '¥x37';
```

【改善策】　Ver.3.40 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 106　sscanf，fscanf，scanf 関数の型指定に関する制限

【内　容】 下記の 1.〜3.の条件を全て満たす場合，3.の型指定に対応する引数の内容が書き換わります。

【 条 件 】

1. sscanf，fscanf，scanf 関数のいずれかを使用している
2. 型指定の数より入力フィールドが少ない
3. 余った最初の型指定文字が s，e，f，g，E，F，G，[ ] のいずれかである【例 1】【例 2】

また下記の 4.〜5.の条件を全て満たす場合，5.の型指定に対応する引数の内容が書き換わります。

【 条 件 】

4. sscanf, fscanf, scanf 関数のいずれかを使用している
5. 型指定文字として[ ] を使用し，[ ]で囲まれた文字パターンが入力フィールドにない【例 3】

【 例 1 】余った最初の型指定文字が「f」の場合

```
char ary1[5];
float  f1 = 2.0, f2 = 3.0;

sscanf ("aaaa", "%s %f %f", ary1, &f1, &f2);
```

← 余った最初の型指定

入力フィールド"aaaa" は「ary1」に文字列として格納されます。しかし「%s」以降の型指定「%f %f」に対応する入力フィールドがありません。このとき，余った最初の型指定に対応する引数「f1」の値が書き換わります。

[期待値]	[出力結果]
ary1 = "aaaa", f1 = 2.0, f2 = 3.0	ary1 = "aaaa", f1 = 0.0, f2 = 3.0

【 例 2 】余った最初の型指定文字が「s」の場合

```
int  data1, data2;
char ary2[5]="test";

sscanf ("1 2", "%d %d %s", &data1, &data2, ary2);
```

← 余った最初の型指定

入力フィールド"1" は「data1」に 10 進整数として格納されます。入力フィールド"2" は「data2」に 10 進整数として格納されます。しかし「%d %d」以降の型指定「%s」に対応する入力フィールドがありません。このとき，余った最初の型指定に対応する引数「ary2」の内容が書き換わります。

[期待値]	[出力結果]
data1 = 1, data2 = 2, ary2 = "test"	data1 = 1, data2 = 2, ary2 = "¥0"

【 例 3 】[ ]で囲まれた文字パターンが入力フィールドにない場合

```
char ary3[5] = "test";
char ary4[5] = "test";
char ary5[5] = "test";

sscanf ("aaaa bbbb cccc", "%s %[a] %s", ary3, ary4, ary5);
```

← 合致しない型指定

入力フィールド"aaaa" は「ary3」に文字列として格納されます。続いて入力フィールド"bbbb"から「a」に合致する文字だけを文字列として「ary4」に格納しようとしますが，合致するパターンがありません。このとき「ary4」の内容が書き換わります。

[期待値]	[出力結果]
ary3 = "aaaa", ary4 = "test", ary5 = "test"	ary3 = "aaaa", ary4 = "¥0", ary5 = "test"

【回避策】 ありません。

【改善策】 Ver.3.40 で修正しました。

No. 107　atoi，atol，strtol，strtoul 関数の引数文字列に関する制限

【内　容】　下記の 1.～5.の条件を全て満たす場合，戻り値が不正になります。また strtol，strtoul 関数の場合は，グローバル変数 errno にマクロ ERANGE が設定されません。

【 条 件 】

1. atoi，atol，strtol，strtoul 関数のいずれかを使用している
2. atoi, atol 関数の場合は，引数文字列を 10 進数値として表現すると，32 ビットを超過する
   strtol, strtoul 関数の場合は，第一引数の文字列を，第三引数により指定した基数の値で表現すると，32 ビットを超過する
3. 2.の引数文字列の先頭からある文字までを変換した数値と，先頭からある文字の次の文字までを変換した数値の絶対値の下位 32 ビット同士を比較して，後者の値が前者の値以上である
4. 3.の比較が文字列の先頭から末尾まで成立する
5. atoi，atol，strtol 関数の場合は，2.の変換数値の下位 32 ビットに符号を付加した値が LONG_MIN～LONG_MAX の範囲内である

【 例 1 】strtoul 関数を使用している場合

```
char *p;
unsigned long ul;

ul = strtoul("123456789", &p, 16);
```

16 進数の 0x123456789 は 32 ビットを超過しています。また，条件 3.の比較が文字列の先頭から末尾まで成立します。
(例) 文字列”12345678”を 16 進数変換した 0x12345678 と，その次の文字を含む”123456789”を 16 進数変換した 0x123456789 の下位 32 ビットを比較して，後者が前者より大きい。
0x12345678 < 0x23456789

[期待値]	[出力結果]
ul = ULONG MAX　errno = ERANGE	ul = 0x23456789

【 例 2 】strtol 関数を使用している場合

```
char *p;
signed long l;

l = strtol("-123456789", &p, 16);
```

16 進数の 0x-123456789 は 32 ビットを超過しています。また，条件 3.の比較が文字列の先頭から末尾まで成立します。

[期待値]	[出力結果]
l = LONG_MIN　errno = ERANGE	l = DCBA9877(-0x23456789)

【 例 3 】atoi 関数を使用している場合

```
signed int i;

i = atoi("5368709120");
```

10 進数の 5368709120(=0x140000000)は 32 ビットを超過しています。また，条件 3.の比較が文字列の先頭から末尾まで成立します。
(例) 文字列” 536870912”を 10 進数変換した 536870912(=0x20000000)と，その次の文字を含む” 5368709120” を 10 進数変換した 5368709120(=0x140000000) の下位 32 ビットを比較して，後者が前者より大きい。
0x20000000 < 0x40000000

[期待値]	[出力結果]
i = LONG MAX	i = 1073741824(0x40000000)

【回避策】　ありません。

【改善策】　Ver.3.40 で修正しました。

No. 108 ビットフィールドをメンバに持つ構造体型変数の初期化の制限

【内　容】 下記の 1.～2.の条件を全て満たす場合，ビットフィールドが正しく初期化されません。【例 1】

【 条 件 】

1. ビットフィールド，構造体（または共用体）の順番で連続しているメンバを持つ構造体型変数を使用している
2. 1.の構造体型変数のメンバを両方とも初期値を利用して初期化している

また下記の 3.～5.の条件を全て満たす場合もビットフィールドが正しく初期化されない場合があります。【例 2】

【 条 件 】

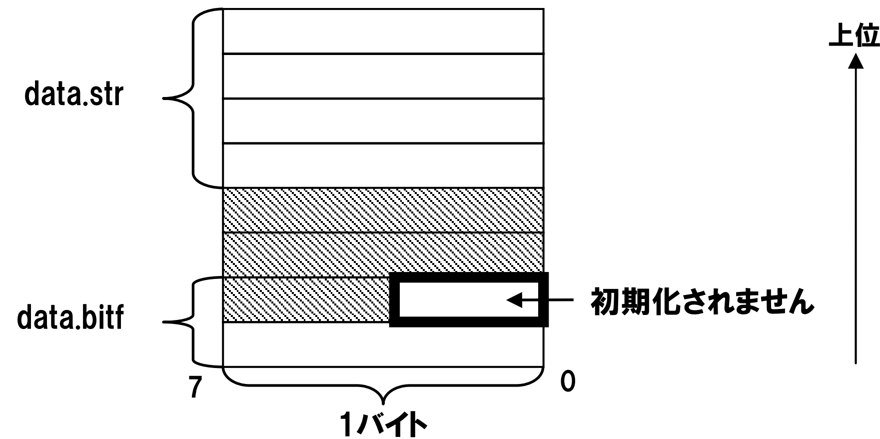
3. 構造体型自動変数の配列を使用している
4. 3.の構造体のメンバに，ビットフィールドと 125 バイト以上の要素を含む
5. 125 バイト以上の要素に対する初期化子を省略し，暗黙の 0 初期化をしている

【 例 1 】

```
struct {
        int bitf : 12 ;         // ビットフィールド
        struct {                // ビットフィールドに続いて構造体
               int s ;
        } str ;
} data = { 0xFFF, { 2 } } ;  // ビットフィールド・構造体のメンバを共に初期化
```

data.bitf の上位 4 ビット分が初期化されません。

[期待値] data.bitf　= 0xFFF	[出力結果] data.bitf　= 0xFF



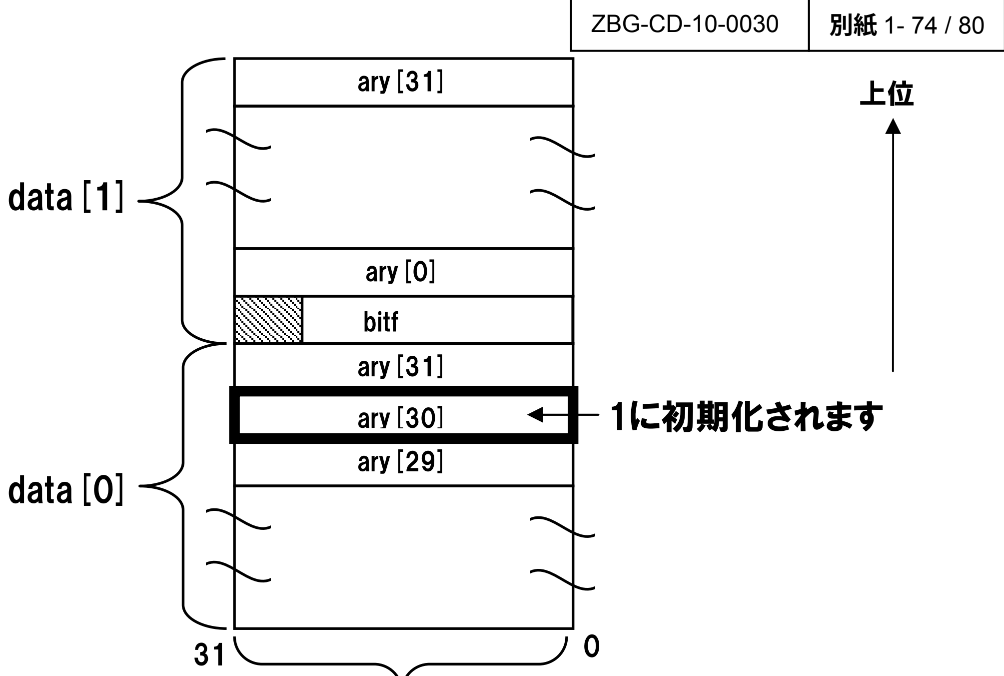
【 例 2 】

```
void func(void)
{
  struct {
      int bitf : 25;                  // ビットフィールド
      int ary[32];                    // 125 バイト以上の要素
  } data[2] = { { 1 }, { 1 } };      // 構造体型自動変数の配列 data を使用
  …                                   // ary を暗黙の 0 初期化
}
```

data[1].bitf を 1 に設定する初期化コードのオフセットが下位方向にずれて，data[0].ary[30] を 1 に初期化してしまいます。data[1].bitf は不定値となります。

[期待値] data[0].ary[30] = 0 data[1].bitf = 1	[出力結果] data[0].ary[30] = 1 data[1].bitf は不定値

【内　容】
（続き）



【回避策】　下記の 1.～2. のいずれかを適用してください。

1. 初期化を代入に変更する

【 例 1 の場合 】

```
struct {
    int bitf : 12 ;
    struct {
            int s ;
    } str ;
} data ;                          // 初期化を使用しない
…
data.bitf = 0xFFF ;               //代入に変更する
data.str.s = 2 ;
```

2. ビットフィールドのメンバを構造体内の構造体にする

【 例 1 の場合 】

```
struct {
    struct {
            int bitf : 12 ;
    } str2 ;                     // 構造体内の構造体にする
    struct {
            int s ;
    } str ;
} data = { { 0xFFF }, { 2 } } ;
```

【改善策】　Ver.3.40 で修正しました。
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 109　条件アセンブル擬似命令のネストの制限

【内　容】　下記の 1.～4.の条件を全て満たす場合，エラーメッセージ（F3510）を出力します。

【 条 件 】

1. .elseif，または.elseifn 擬似命令を使用している
2. 1.の擬似命令で指定した式が真である
3. 1.の擬似命令に対応する擬似命令が.elseif，または.else 擬似命令である
4. 1.の擬似命令に対応するブロック間に条件アセンブル疑似命令をネストしている

【 例 】

```
.set  FLAG1, 0
.set  FLAG2, 1
.set  FLAG3, 1
.set  FLAG4, 0

.if FLAG1 == 1
  .set   TEMP, 1
.elseif FLAG2 == 1  --条件 1./条件 2.に合致
   .if FLAG3 == 1    --条件 4.に合致    ┐
     .set   TEMP, 2                     ├ ネスト部
   .endif                               ┘
.elseif FLAG4 == 1  --条件 3.に合致
  .set   TEMP, 3
.endif
```

【回避策】　条件アセンブル擬似命令をネストする際には，下記の 1.～2. のいずれかのブロックで行ってください。

1. .if 擬似命令のブロック

```
.if FLAG1 == 1
  .set   TEMP, 1
.else
  .if FLAG2 == 1
    .if FLAG3 == 1      ┐
      .set   TEMP, 2    ├ ネスト部
    .endif              ┘
  .elseif FLAG4 == 1
    .set   TEMP, 3
  .endif
.endif
```

2. .elseif～.endif 擬似命令のブロック

```
.if FLAG1 == 1
  .set   TEMP, 1
.elseif FLAG4 == 1
  .set   TEMP, 3
.elseif FLAG2 == 1
   .if FLAG3 == 1     ┐
     .set   TEMP, 2   ├ ネスト部
   .endif             ┘
.endif
```

【改善策】　Ver.3.40 で修正しました。

No. 110　switch，if 文内での代入の制限

【内　容】 下記の 1.～2.の条件を全て満たす場合，その後の ca850 の最適化により，switch 文あるいは if 文内の処理が不正となる場合があります。

【 条 件 】

1. C ソース上において(ア)～(ウ)の条件を全て満たす【例 1】

(ア)　Ver.2.50 未満の場合，コンパイラの最適化オプションとして"-Os" "-Ot" のいずれかと，"-Ol"を指定しており，"-Wi,-O4" は未指定である
Ver.2.50 以上の場合，コンパイラの最適化オプションとして"-Os" "-Ot" のいずれかを指定しており，"-Wi,-O4" は未指定である

(イ)　switch 文，あるいは if 文内で同じ変数に値を代入する処理が並列している

(ウ)　(イ)の代入後，同じ位置に分岐する

【 例 1 】

```
int func(int val) {
  int res;
  switch (val) {
  case 0xA: res = 0x1; break;
  case 0xB: res = 0x2; break;
  case 0xC: res = 0x3; break;
  case 0xD: res = 0x3; break;      ← 同じ変数"res"に値を代入する処理が並列
  case 0xE: res = 0x3; break;
  default:  res = 0x3; break;
  }
  return res;                      ← 変数"res"に値を代入後，同じ位置に分岐
}
```

2.　1.の C ソース記述により出力されたアセンブラが(エ)～(ク)の条件を全て満たす【例 2】

(エ)　2 つ以上の基本ブロック(A)の末尾で，同じレジスタへ mov 命令または ld 命令で転送する

(オ)　(A)のブロックが全て１つのブロック(B)へ合流する

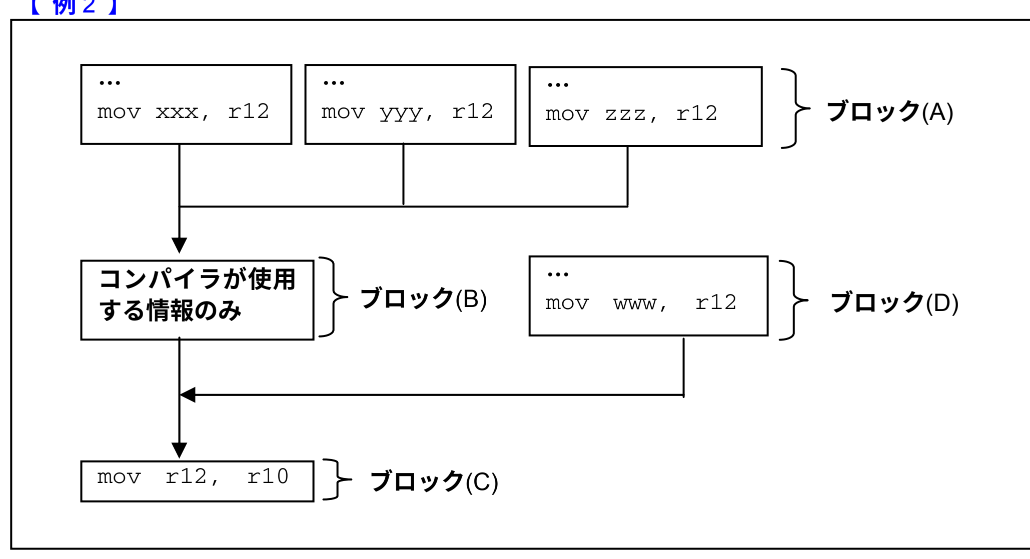
(カ)　(B)のブロックに命令がなく，出力コードとはならないコンパイラが使用する情報が１つ以上含まれる

(キ)　(B)の次のブロック(C)で，(A)で転送したレジスタを他のレジスタへ転送する

(ク)　ブロック(C)に直接合流するブロックの内，(A)と同様のレジスタへ転送を行っているブロック(D)が存在する

(注意)　「基本ブロック」とは，必ず先頭の命令から入り，分岐命令までの命令並びです。

【 例 2 】



【回避策】 **下記の** 1.〜3. **のいずれかを適用してください。**

1. __asm("¥n"); **を挿入する（挿入が必要な位置候補をチェックツールが出力します）**

```
int func(int val) {
  int res;
  switch (val) {
  case 0xA: res = 0x1; break;
  case 0xB: res = 0x2; break;
  case 0xC: res = 0x3; break;
  case 0xD: res = 0x3; break;
  case 0xE: res = 0x3; break;
  default:  res = 0x3; break;
  }
  __asm("¥n"); ←———————__asm("¥n"); の挿入
  return res;
}
```

2. **最適化オプションとして**”-O” **以下を指定する**
3. -Wi,-O4 **オプションを指定する**

【改善策】 Ver.3.40 **で修正しました。**
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 111　キャストを伴う比較演算が不正に最適化される制限

【内　容】　下記の 1.～5. の条件を全て満たす場合に，1.の比較演算を不正に最適化します。

【 条 件 】

1. 整数型同士の比較演算(<, <=, >, >=, ==, !=)を行っている
2. 比較の一方が下記のいずれかのキャストを行っている
   - signed char -> unsigned char
   - signed char -> unsigned short
   - unsigned char -> signed char
   - signed short -> unsigned short
   - unsigned short -> signed short
   - unsigend short -> signed char
3. キャスト後の型が，比較のもう一方の型に等しい
4. キャストを行っていないもう一方が，最適化の影響で定数として扱われる
5. 4.の定数と 2.のキャストを無効とした場合の 1.の比較が，常に真または偽となる

【 例 】

```
short s; unsigned short us;
s = -1;
…
if (s == (short)us)
```

4 行目の比較演算において，左辺の"s"が CA850 の最適化の影響で定数"-1"として扱われた場合に，右辺の"us"の short 型へのキャストを無効とした場合の比較結果は常に偽となります。そのため，1.～5. の全ての条件に該当します。この比較演算は不正に最適化されて，削除されます。

【回避策】　制限に該当する場合，不正に最適化される比較演算の行番号をチェックツールが出力します。
不正に最適化される比較演算が記載されている関数がインライン展開される場合，その関数をコールしている行番号を出力します。
不正最適化される比較演算の直前に「__asm("¥n");」を挿入してください。

【 例 】

```
short s; unsigned short us;
s = -1;
…
__asm("¥n");
if (s == (short)us)
```

【改善策】　Ver.3.46 で修正する予定です。
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

No. 112　代入文に対する不正移動の制限

【内　容】　下記の 1.～2. の条件を全て満たす場合に，基本ブロック 1 内の変数 X への代入文を，基本ブロック 2 内の変数 X への代入文の後に不正に移動します。

【 条 件 】

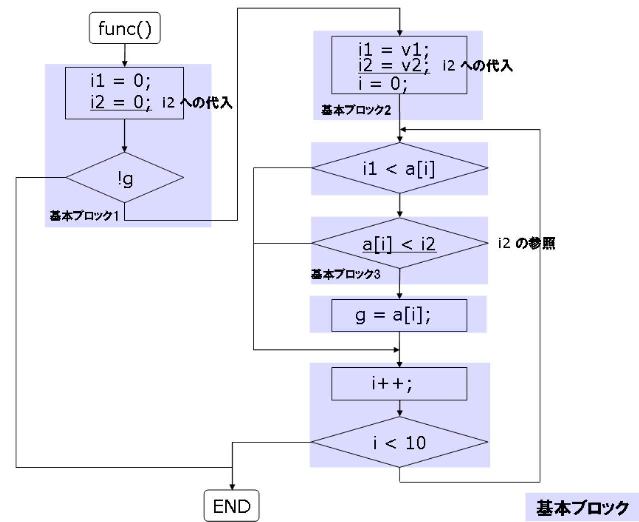
1. 最適化レベルとして"-Og" "-O" "-Os" "-Ot"のいずれかを指定している
2. 関数の制御フローが(a)～(e)のような基本ブロックから構成される
   - (a) volatile 指定されていない自動変数，あるいは volatile 指定されていない引数 X に対して，基本ブロック 1 と基本ブロック 2 で値を代入し，基本ブロック 3 で参照する
   - (b) 基本ブロック 2 の X への代入値は，volatile 指定された変数式，あるいは周辺 I/O レジスタ名を使用した式である
   - (c) 基本ブロック 2 に合流するのは，基本ブロック 1 に限定される
   - (d) 基本ブロック 2 の後に，基本ブロック 3 に合流するパスと合流しないパスが存在する
   - (e) 基本ブロック 1 から基本ブロック 3 の間に X への間接アクセスが無い

(注意)　「基本ブロック」とは，一つの入り口（すなわち、内部のコードが他のコードの分岐先になっていない)と一つの出口を持ち、内部に分岐を含まないコードを指します。

【 例 】

```
int g, a[10];
volatile int v1, v2;
void func(void) {
  int i1 = 0, i2 = 0;
  if(!g){
    int i;
    i1 = v1;
    i2 = v2;
    for(i = 0; i < 10; i++)
      if(i1 < a[i] && a[i] < i2)
        g = a[i];
  }
}
```

上記の例は以下のような基本ブロックから構成される制御フローとなり，(a)～(e)の条件に該当します。



【回避策】 **制限に該当する場合，チェックツールが行番号情報を出力します。**
その行番号の直前に「__asm(“ ¥n ”);」を挿入してください。

【 例 】

```
if(!g){
  int i;
  i1 = v1;
  i2 = v2;
  __asm("¥n");
  for(i = 0; i < 10; i++)
    if(i1 < a[i] && a[i] < i2)
      g = a[i];
}
```

【改善策】 Ver.3.46 **で修正する予定です。**
また，本制限に該当するかをチェックするツールを用意しています。
チェックツールに関しましては，弊社営業または特約店にお問い合わせください。

その他パッケージ・ツールの制限事項一覧

1. 製品履歴

No.	仕様変更・追加／制限事項	バージョン									
		V2.41	V2.50	V2.6x	V2.70	V2.72	V3.00	V3.10	V3.20	V3.30	V3.4x
1	PM+ 正規表現を用いた置換の制限	×	×	×	○	○	○	○	○	○	○
2	PM+ かぎ括弧文字の制限	×	×	×	○	○	○	○	○	○	○
3	PM+ .prw ファイルの関連付けの制限	−	−	−	×	×	×	○	○	○	○
4	stk850 システム・コール（RX850，RX850Pro）の表示の制限	−	−	−	−	−	×	○	○	○	○
5	stk850 システム・コール（RX850）のスタックサイズの制限	−	−	−	−	−	×	○	○	○	○
6	PM+ コマンドファイルの制限	−	×	×	×	×	×	○	○	○	○
7	PM+ ライブラリからのシステム・コール（RX850，RX850Pro）リンクの制限	−	−	−	−	−	×	○	○	○	○
8	PM+ インクルードファイルの制限	−	−	−	×	×	×	×	○	○	○
9	PM+ C ソースのインクルードファイル情報取得の制限	−	−	−	−	−	−	−	×	×	○

×：該当する
○：該当しない
−：対象外

2. 使用制限事項の詳細

No. 1 PM+ 正規表現を用いた置換の制限

【内　容】 PM+ にて，文字列の置換の際に，正規表現の行頭を表す“ ^ ”，行末を現す“ $ ”を使用して置換処理を行った場合，該当文字列の前や後ろに存在する改行コードを含めて，置換指定文字列と置き換えてしまいます。

（例）文字列の置換ダイアログで，「正規表現」を使用するチェックボックスをオンにし，以下のようなテキストに対し

```
------------------
abc
xyz
abc
------------------
```

置換前文字列を ” ^xyz ”，置換後文字列を ” ABC ” とすると

```
------------------
abcABC
abc
------------------
```

となり，改行コードが一緒に置換されてしまいます。

【回避策】 置換の際には，正規表現を使用しないでください。

【改善策】 PM+ Ver.5.20（CA850 Ver.2.70 に同梱）で修正しました。

No. 2 PM+ かぎ括弧文字の制限

【内　容】 PM+ にて，ファイル名，パス名，プロジェクト・グループ名にかぎ括弧（“ [”，または“] ”）を使用すると，ワークスペースの読み込みや，個別オプションの設定ができません。

【回避策】 ファイル名，パス名，プロジェクト・グループ名にかぎ括弧（“ [”，または“] ”）を使用しないでください。

【改善策】 PM+ Ver.5.20（CA850 Ver.2.70 に同梱）で修正しました。

No. 3　PM+ .prw ファイルの関連付けの制限

【内　容】 .prw ファイルを PM+に関連付けしていますが，空白を含むフォルダ名のワークスペース・ファイルをダブルクリックすると正常にワークスペースを開けない場合があります。

【回避策】 PM+の "ワークスペースを開く(W)" メニューで，ワークスペース・ファイルを開いてください。

【改善策】 PM+ Ver.6.10 で修正しました。

No. 4　stk850 システム・コール（RX850，RX850Pro）の表示の制限

【内　容】 RX850，RX850Pro のシステム・コールのスタックサイズが不明になり，「？」と表示されてしまいます。

RX850 Ver.3.20，RX850Pro Ver.3.20 のシステム・コールのスタックサイズは全て 0 ですので，スタックの計算値には影響はありません。

【回避策】 RX850 Ver.3.20，RX850Pro Ver.3.20 のシステム・コールについては，スタックサイズ不明のままご使用いただくか，スタックの加算サイズを 0 にしてください。

【改善策】 stk850 Ver.2.11 で修正しました。

No. 5　stk850 システム・コール（RX850）のスタックサイズの制限

【内　容】 RX850 Ver.3.20 の次のシステム・コールのスタックサイズが，本来は 0 ですが，12 加算されてしまう可能性があります。

- ter_tsk
- rel_wai
- get_blk
- pget_blk
- tget_blk
- rel_blk

制限事項 No.4 に該当している場合には，システム・コールのスタックサイズが不明になり，0 で計算されますので本制限事項には該当いたしません。

stk850 で上記システム・コールのスタックの加算サイズを指定していないのに，「サイズ不明関数リスト」に登録されていない場合に，該当いたします。

【回避策】 該当するシステム・コールのスタックの加算サイズを 0 に変更してください。

【改善策】 stk850 Ver.2.11 で修正しました。

No. 6　PM+ コマンドファイルの制限

【内　容】 コンパイラのオプションの設定，または，アセンブラのオプションの設定で，"コマンドファイルの作成" のチェックをしている場合に，Project Window でソース・ファイルを選択して "コンパイル" をすると，コマンドファイルが削除されてしまい，次のビルド，または，リビルド時にエラーになります。

```
F1303: cannot open file 'xxxx.cca'

F3503: can not open file xxxx.cas
```

【回避策】 PM+の[プロジェクト]-[メイクファイルのエクスポート]メニューを選択して，ビルドに必要なファイルを再生成させてください。

【改善策】 CA850 Ver.3.10，PM+ Ver.6.11 で修正しました。

No. 7　PM+ ライブラリからのシステム・コール（RX850，RX850Pro）リンクの制限

【内　容】　下記の条件全てを満たす場合に，RX850，RX850Pro のシステム・コールがリンクエラーになります。

```
F4452: undefined symbol.
```

【 条 件 】

1. RX850，RX850Pro を使用している
2. リンカオプションの設定でライブラリを指定している
3. 2 で指定したライブラリでシステム・コールが使用されている
4. 3 のシステム・コールがソース・ファイル登録したプログラムでは使用されていない

【回避策】　次の方法で回避してください。

1. リンカオプションの設定の“ライブラリ”タブの“ライブラリ[-l]”に RX850，RX850Pro のライブラリを指定
   【 例 】
   RX850 Ver3.20 を使用していて，ユーザのライブラリ名が libusr.a の場合
   ```
   usr;rx
   ```
2. リンカオプションの設定の“ライブラリ”タブの“ライブラリのパス[-L]”に RX850，RX850Pro のライブラリ・パスを指定
   【 例 】
   RX850Pro Ver3.20 を使用している場合
   ```
   C:¥Program Files¥NEC Electronics Tools¥RX850 Pro¥V3.20¥lib850e¥r32
   ```
3. RTOS[RX850Pro]の詳細設定の，次の 2 つのチェックボックスを解除【RX850Pro のみ】
   - ニュークリアス・ライブラリをリンクする(U)
   - インタフェース・ライブラリをリンクする(R)

【改善策】　CA850 Ver.3.10，PM+ Ver.6.11 で修正しました。

No. 8　PM+ インクルードファイルの制限

【内　容】　ソースファイル上のインクルードファイルの指定の中で空白を含むファイル名が存在すると，そのインクルードファイルと，その後に記述された全てのインクルードファイルの扱いが，次のようになります。

- ProjectWindow のインクルードファイルにファイル登録されない
- メイクファイルに登録されない
- ビルド時の依存関係の対象にならない

条件コンパイルが偽の部分のインクルードファイルも該当します。

PM+でのソースファイル上のインクルードファイルの解析部分のみ該当し，実際のコンパイルの際の処理は該当しません。

【 例 】

```
#include “C:¥i n c¥test.h”
```

【回避策】　空白を含めないような指定にしてください。

【改善策】　CA850 Ver.3.20，PM+ Ver.6.31 で修正しました。

No. 9　PM+ C ソースのインクルードファイル情報取得の制限

【内　容】　以下の 1.または 2.のいずれかを満たしている場合，

1. ソースファイルの先頭から#include の前の「"」の個数が奇数個である場合
2. ソースファイルの先頭から#include の前に「<」を使用しており，かつ，その「<」の後に「>」がない場合

#include 記述によりインクルードしたヘッダファイルの扱いが，以下のようになる可能性があります。ただしリビルド時の処理には影響しません。

- ProjectWindow のインクルードファイルにファイル登録されない
- メイクファイルに登録されない
- ビルド時の依存関係の対象にならない

【 例 1 】

```
/*xxx""xxx*/
#include "usr1.h"   ← #include の前の「"」の個数が 3
/*xxx"xxx*/
#include "usr2.h"  ← #include の前の「"」の個数が 6
```

usr1.h が本制限に該当します。また usr1.h 内でインクルードしているヘッダファイルも本制限に該当します。usr2.h は本制限に該当しません。

【 例 2 】

```
/*xxxx><xxx*/
#include <usr3.h> ← #include の前に「<」を使用しているが，
                    「<」の後ろに「>」がない
#include <usr4.h>  ← #include の前に「<」を使用しているが，
                    「<」の後ろに「>」がある
```

usr3.h が本制限に該当します。また usr3.h 内でインクルードしているヘッダファイルも本制限に該当します。usr4.h は本制限に該当しません。

【回避策】　以下の 1.または 2.のいずれかを適用してください。

1. ソースファイルの先頭から#include の前の「"」の個数が奇数個である場合は，偶数個になるように「"」を追記する
2. ソースファイルの先頭から#include の前に「<」を使用しており，その「<」の後に「>」がない場合は，「>」を追記する

【改善策】　CA850 Ver.3.40 で修正しました。

以上